ACE Link - An Approach to Integrating Command and Control Model Architectures

Final Report

Prepared By Modasco, Inc.

For Air Force Research Laboratory
Contract F33615-00-C-1669

Unclassified

ACE Link 21 November 2000

REPORT DOCUMENTATION PAGE

Form Approved
OMB NO. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE 01 DEC 00	3. REPORT TYPE AND DATES COVERED Final Report: JAN 00 - DEC 00

4. TITLE AND SUBTITLE
ACE LINK - An Approach to Integrating Command and Control Model Architecture

5. FUNDING NUMBERS
C
F33615-00-C-1669

6. AUTHOR(S)
Woodring, John W.; Kashmiri, Rafiah
Navarro, Guillermo

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Modasco, Inc.
4303 Vineland Road, Suite F-7
Orlando, FL 32811

8. PERFORMING ORGANIZATION REPORT NUMBER
FR1

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
U. S. Army Research Office
P.O. Box 12211
Research Triangle Park, NC 27709-2211

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
USAF/AFMC
AIR FORCE RESEARCH LAB
2310 EIGHT STREET, BUILDING 167
WRIGHT-PATTERSON AFB, OHIO
45433-7801

11. SUPPLEMENTARY NOTES
The views, opinions and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy or decision, unless so designated by other documentation.

12 a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12 b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

Report developed under SBIR contract for topic AF00-116.

A software package that provides powerful tools to assist engineers in designing and analyzing complex systems is proposed. This package allows multiple designers, working independently in a collaborative environment, to create a digital design of a process in individual modules, bind them together and simulate their performance under internal and external control. The resulting virtual design is tied to a database of system requirements and automatically responds to requirement changes. System designs are developed as Colored Petri Nets using a graphical editor supported with tools to create, modify, edit, store and merge designs. Other tools help the designer describe the system's behavior and relate it to both objects defined and stored in a database and to external modules from legacy or ancestral simulations. Analysis and evaluation tools are provided to execute large numbers of test scenarios, measure system-performance metrics and identify and report failures and malfunctions. This methodology is robust with respect to describing concurrent events arising from multiple external controls and accurately models the behavior of distributed-processing systems. The design tools have the power and generality to model and simulate virtually all commercial and military processes, including those related to human performance in a combat environment.

14. SUBJECT TERMS
SBIR Report
Modeling and Simulation; Business Engineering;
Process Modeling; Command and Control Architecture;
Collaborative Virtual Prototyping

15. NUMBER OF PAGES
131

16. PRICE CODE

17. SECURITY CLASSIFICATION OR REPORT	18. SECURITY CLASSIFICATION ON THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

TABLE OF CONTENTS

Section I

Presentation Overview

This presentation is divided into four parts:

- **ACE Link - Description of the Opportunity and Approach**
- **ACE Link Interface Demonstration**
- **ACE Link Application to THUNDER**
- **THUNDER Demonstration**

ACE Link Development Team Members & Their Respective Responsibilities

Modasco, Inc.

- Rafiah Kashmiri - Program Management
- Dr. John Woodring – Principle Investigator for ACE Link Architecture

Emergent Technology, Inc.

- Greg Jablunovsky - THUNDER Integration

The Opportunity

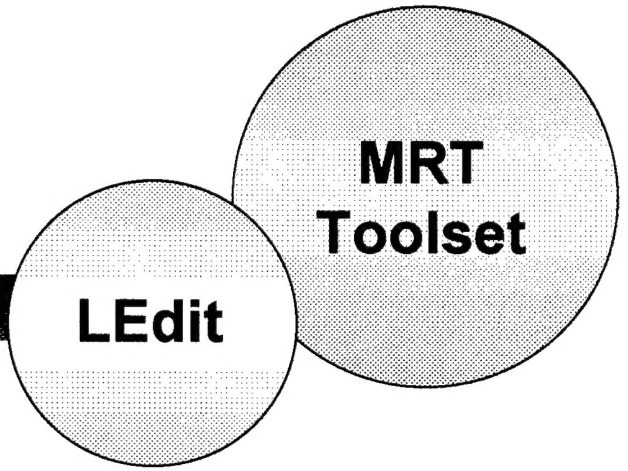
The Air Force has developed powerful tools for designing complex systems, but there is no direct way of evaluating the military worth of a proposed design.

Architecture Design Tool = LEdit

Combat Evaluation Model = THUNDER

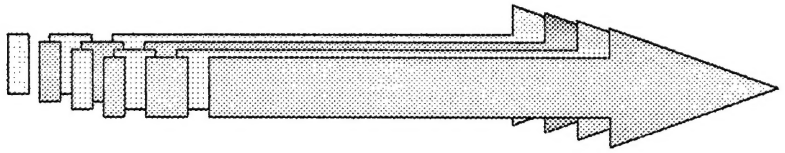
ACE Link = Architecture + Combat Evaluation

LEdit



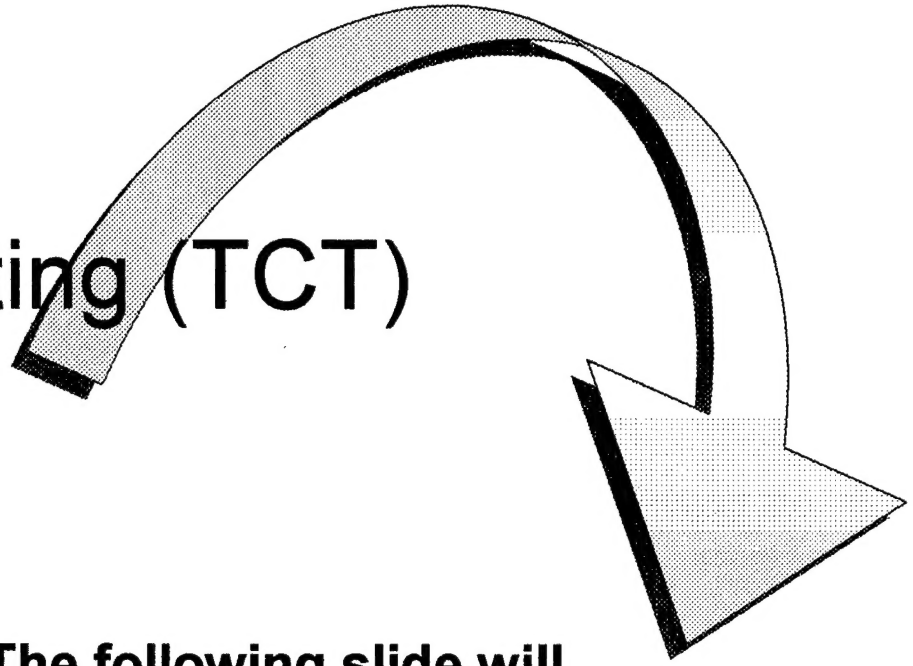
LEdit is a part of the MRT Toolset used to graphically describe Command and Control architectures

- It uses Colored Petri Nets as its design language

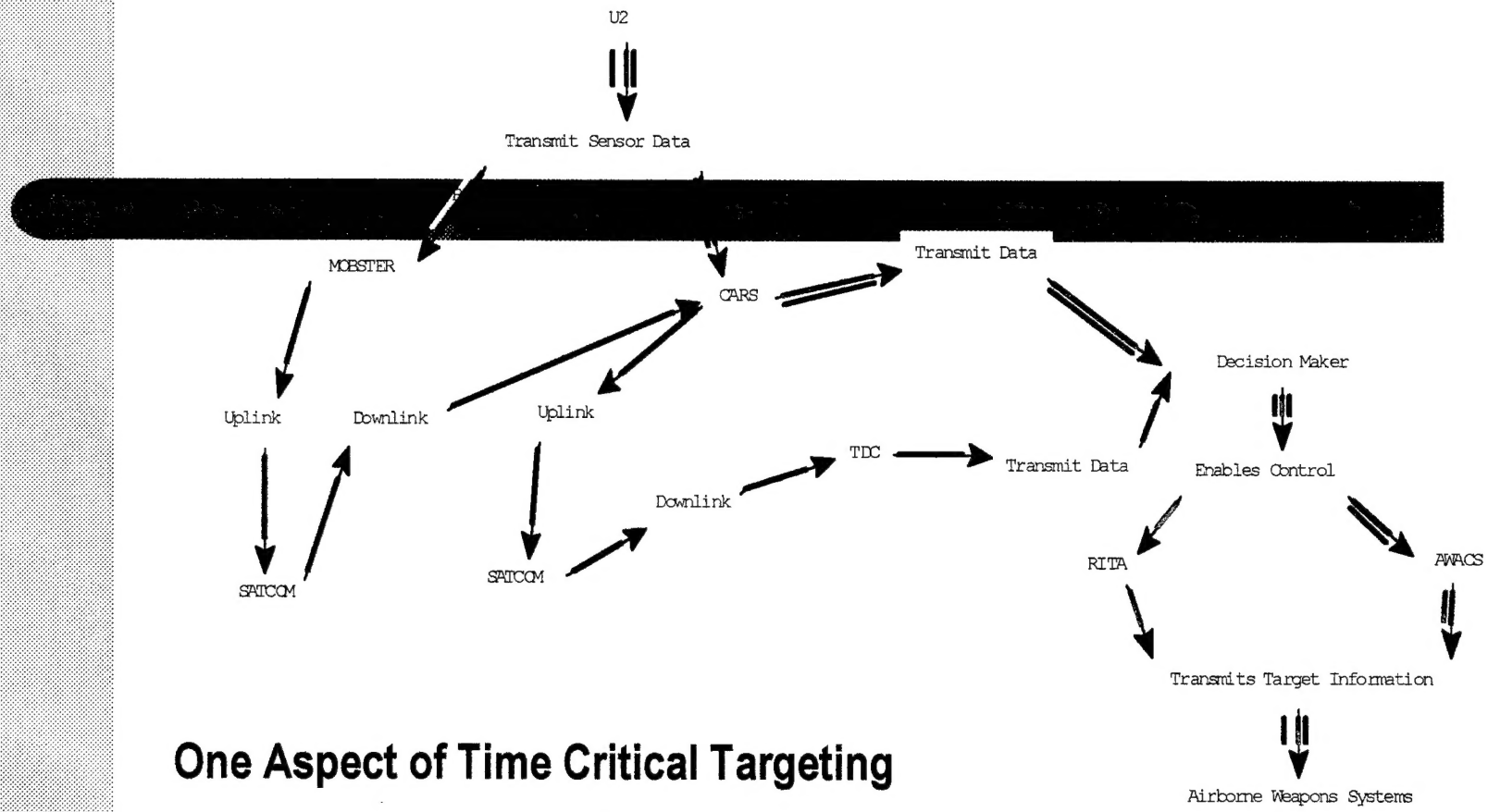


- While simple to use and powerful in its ability to clarify a complex system, its use has been, up until this time, limited to **qualitative** analysis.

Time Critical Targeting (TCT)



The following slide will illustrate how one aspect of TCT can be modeled using LEdit.

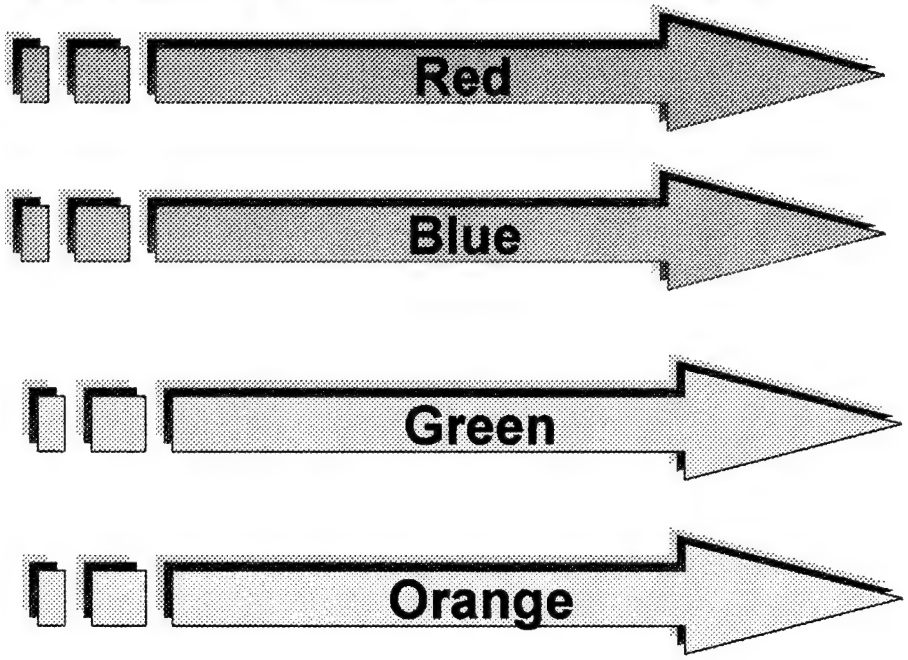


One Aspect of Time Critical Targeting

(Network Designed Using LEdit)

What does this graphical design illustrate?

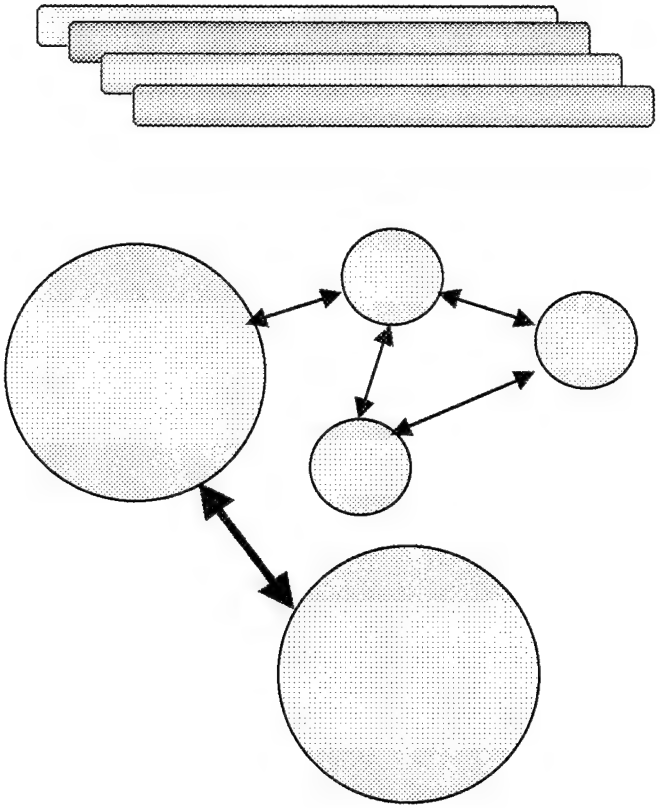
The various communications paths (there are four) between a U2 and the Shooter.



Each path is distinguished by color

What Does the Network Graph Tell You?

- That there are four possible communications links between the sensor (U2) and the Shooter (F-15).
- For any single communications link to be operable, all of its intermediate communications nodes must also be operable.



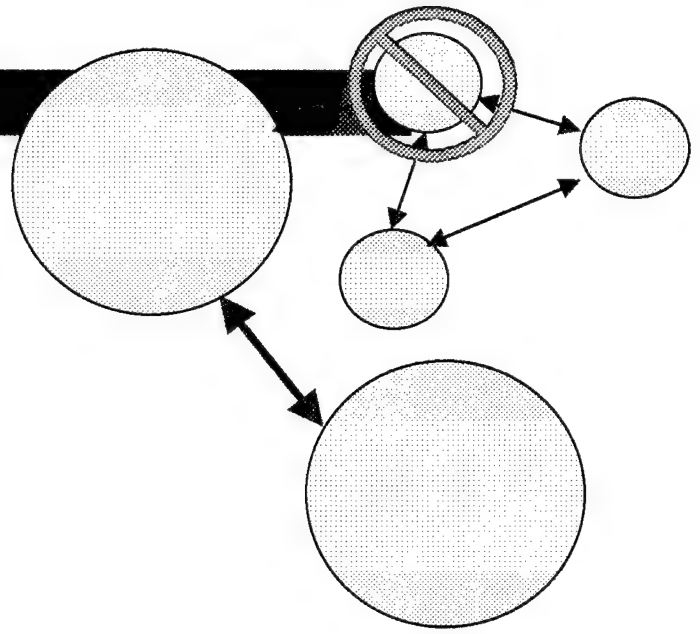
THUNDER

There is no specification of individual communications nodes or communications links. Therefore, the model is not sensitive to realistic events such as:

THUNDER is a simulation of theater warfare used by modelers to quantitatively determine how and why various mission outcomes occur. It models Time Critical Targeting in an unrealistic and oversimplified way.

- A node that has not yet been established in the event
- Time delays that are specific to the individual nodes
- A node that has been partially or totally destroyed

What Does the Network Graph **Fail** to Tell You?



Which is the optimal path based upon the minimal communications time delay if:

- All the communications nodes are operable
- One or more nodes is inoperable

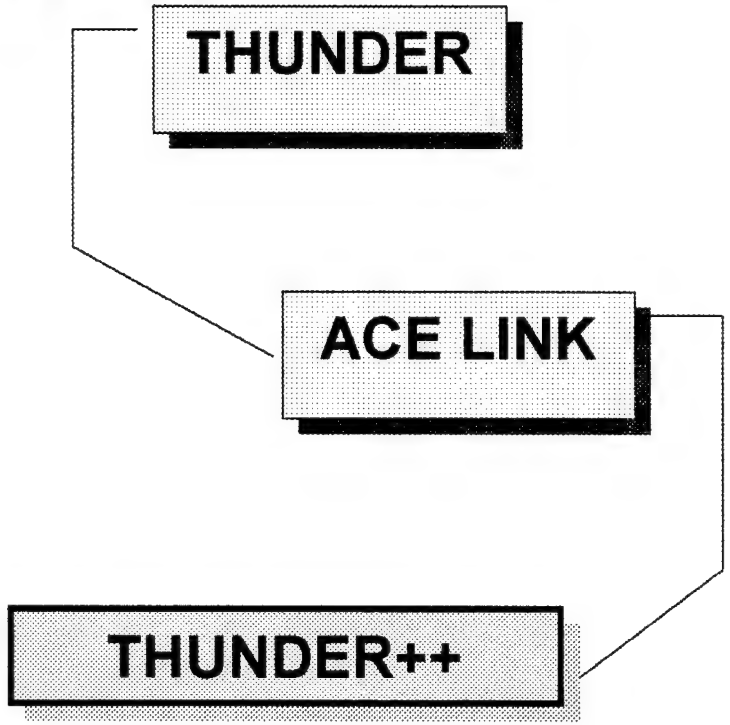
This is the current limitation of the usefulness of LEdit

The Problem:

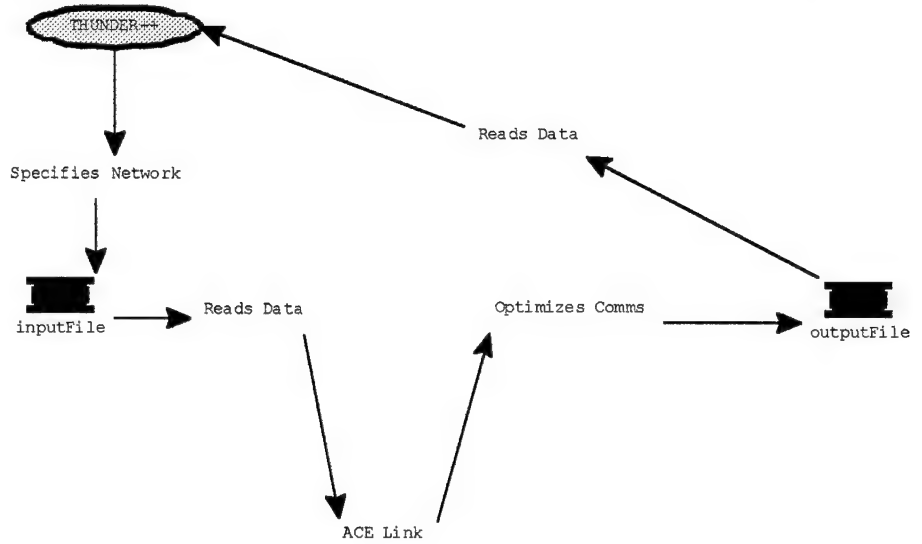
How to integrate the graphical design capability of LEdit with the simulation capability of THUNDER?

ACE Link - A Prototype Solution

ACE Link is Middleware that sits between LEdit and THUNDER and Creates a Synthesis of the Two Tools



High Level ACE Link Structure



How Do We Implement ACE Link?

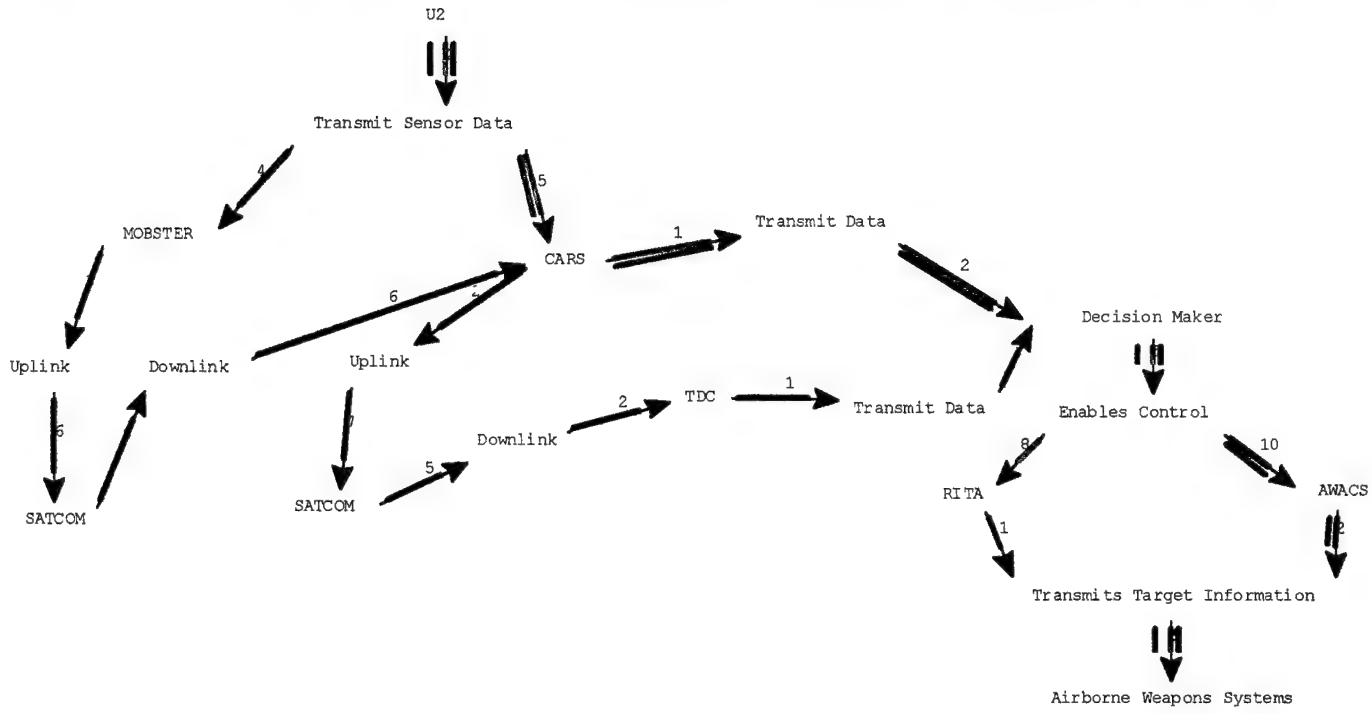
Step 1: Embed Time Delays in LEdit Design

Open LEdit Graph

- Double Click on Each Path Segment
- Enter the time delay for each communications-path segment in the "Edit edge" Dialog Box
- Click on the "OK" Button

U2-Shooter - LEdit Design with Embedded Time Delays

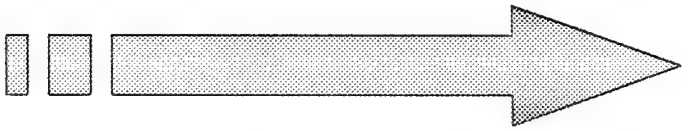
(Note: LEdit Often Superimposes Text and the Path Segment!)



There Are Four Communications Links (1-2):

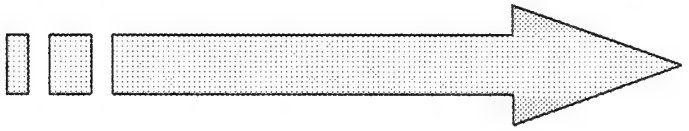
Loop 1 (Red):

U2→CARS → DECISION MAKER → AWACS →
Airborne Weapons System = 36 sec.



Loop 2 (Green):

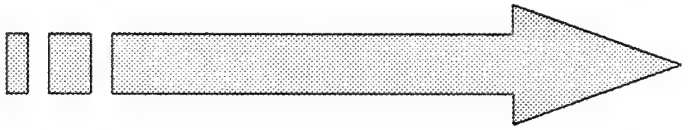
U2→CARS → DECISION MAKER → RITA →
Airborne Weapons System = 33 sec.

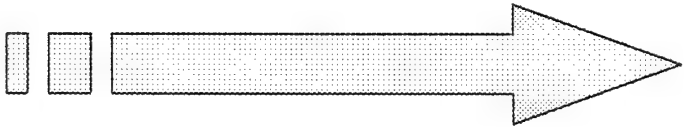


There Are Four Communications Links (3-4):

Loop 3 (Blue):

U2→MOBSTER →SATCOM →CARS →SATCOM →TDC →
DECISION MAKER → AWACS → Airborne Weapons System = 70 sec.





Step 2: Constructing the Input File

(This function is performed by THUNDER)

LEdit File Name:
\tct_2.edt
Transmitter Node:
U2
Receiver Node:
Airborne Weapons System
Non-Operational Nodes:

Step 3 - Execute ACE Link

Double Click on the ACE Link Icon - Ace Link then:

- Opens the LEdit File Specified in Line 2 of the Input File
- Reads the Name of the "Transmitter" Node Specified in Line 4 of the Input File
- Reads the Name of the "Receiver" Node Specified in Line 6 of the Input File
- Reads the Names of the "Inoperable" Nodes on Line 8.. (None in this case)
- Finds the Communications Paths With the Minimum Time Delay Subject to These Conditions
- Creates the Output File for THUNDER to Read

(Enter 0 as required to terminate ACE Link Execution)

Step 4 - Read the File Created by ACE Link

ACE Link Has Correctly Identified the Optimum Path (Green Loop), the Node Sequence For This Path and the Delay Time of 33 Seconds.

Minimum Time Delay = 33
U2
CARS
Decision Maker
RITA
Airborne Weapons Systems

Which Communications Path Would Be Optimal if RITA Was Non-Operable?

Enter RITA as a Non-Operable Node in the Input File and Execute ACE Link:

Input File	Output File
LEdit File Name: \tct_2.edt Transmitter Node: U2 Receiver Node: Airborne Weapons Systems Non-Operational Nodes: RITA	Minimum Time Delay = 36 U2 CARS Decision Maker AWACS Airborne Weapons Systems

The Significance of ACE Link is Related to its Ability to Quantify A System Design.

Prior to ACE Link, LEdit was a One-Dimensional Tool

- It provided only a qualitative picture of the system
- Data Describing the System Was Not Encapsulated in the Design
- Analysis of the Design Had to Be Performed "Elsewhere" - in other Combat Simulation Models Such as THUNDER.

The Result: It is Virtually Impossible to Re-Use an LEdit Design in a New Model

How is ACE Link Related to Object Oriented Analysis & Design?

The Modern View of Simulation and Modeling is Based Upon the Ideas of OOAD. The Model Should:

- Be Designed Visually

  Example: CPN Model Created by LEdit

- Encapsulate Data Describing the Design

  Example: Communications Time Delays

- Provide Interactive Services to Answer Questions About the System

  Example: What is the Optimum Path Subject To a Set of Constraints.

What Are the Advantages of This Approach?

- Once a System Has Been Designed, It Can Be Implemented In a Number of Different Simulation Models (THUNDER, STORM..)
- Modification of the System Is Simplified, Since the Attributes and Methods of the System are Encapsulated In It.
- Directly Supports Object Oriented Software Development
- Directly Supports RDB Interfaces

The Problem

No traceable connection between C2 Architecture & Military Worth

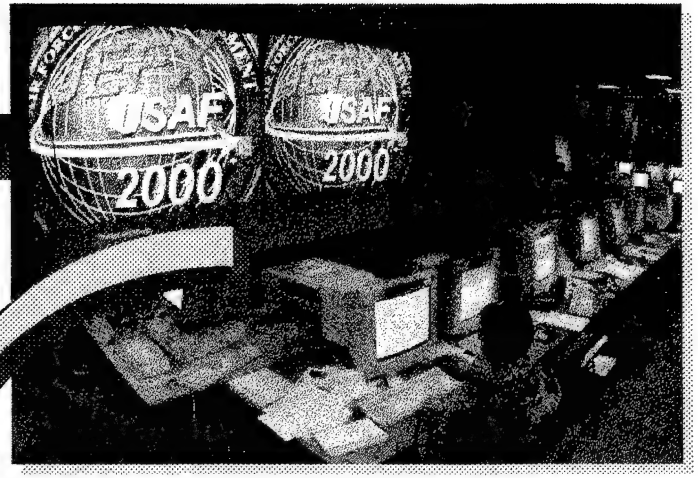
- **Persistent difficulty in Command & Control (C2) to effectively relate performance to effectiveness**
  - Military Operations Research Society C4ISR workshops
  - National Defense University
  - ESC

Problem Domain

- **Time-Critical-Targeting (TCT)**
  - The sensor-decision maker-shooter loop is an active area of C2 experimentation in AF and Joint domains
  - This mission areas has a critical bearing on decisive operations and

Opportunity

Bridge gap between existing tools with complimentary analytic capability

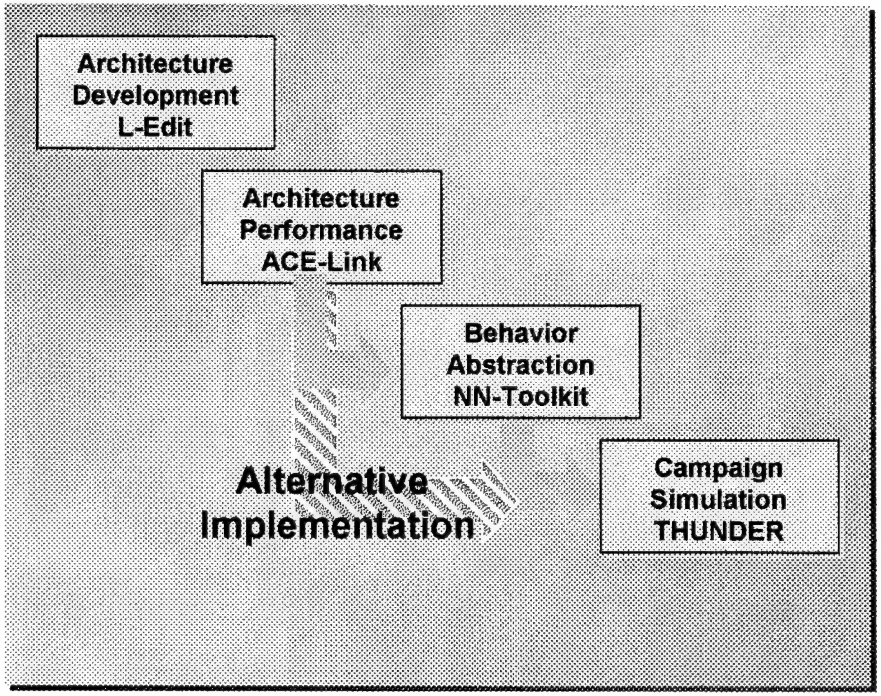


- **L-Edit has a growing user base in core C2 operational and system organizations for Architecture Analysis**
- **THUNDER is the AF Suite of Models Campaign Analysis tool**

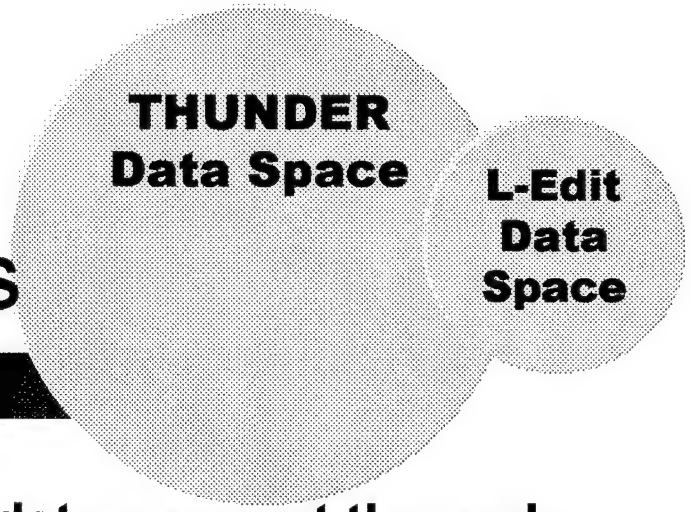
Approach

Prove Concept with limited but useful prototype

- **ACE-Link Application provides a "thin", adaptable interface that integrates the analytic power of MRT and THUNDER**



ACE-Link Connects to Intersection of Data Sets

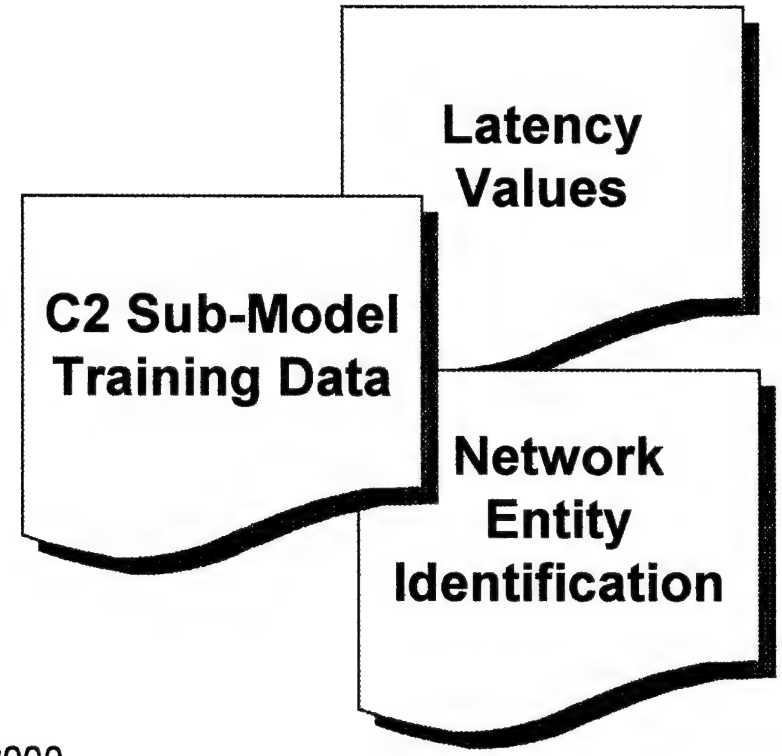


- **Ace-Link interfaces L-Edit to THUNDER data space at the node level**
- **Links and link latencies are outside the THUNDER data space**
  - Derived from published sensor-shooter studies from OSD/C3I, Air Combat Command Data and studies, live experiments like JEFX, and "operator estimates" from the Air and Space C2ISR Center
- **Entity data definitions are outside the L-Edit Data Space**
  - Doesn't know "who" they're fighting for or against
  - Doesn't know "what" satellite or AWACS
  - Doesn't know "where" entities are located
  - Doesn't know "when" they deploy or start fight

Data Interface

ACE-Link Provides C2 Architecture and Performance Data

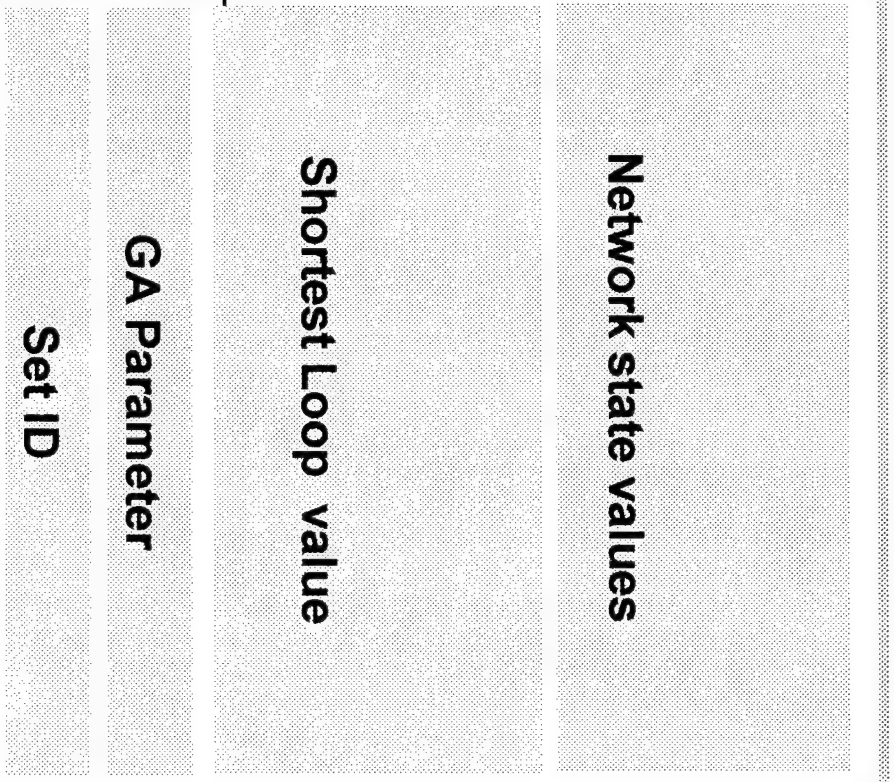
- **Strictly formatted files with:**
  - Connectivity data encoded in state-performance pairs
  - Performance of alternative "Loops"
  - Description of network nodes as entities in THUNDER data space



C2 Sub-Model Data

State-Performance Relationships Are the Key to Moving Architecture Information Between the Tools

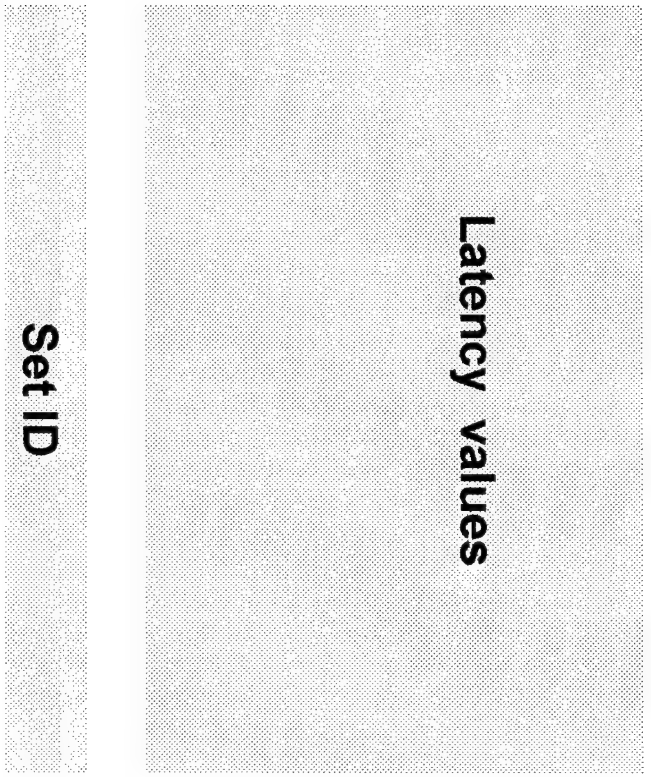
# of Input-Output pairs
Number of Inputs
Number of Outputs



- ACE-Link automates generation from L-Edit data
- ASCII File with strict structure defining sampling of pairs showing architecture connectivity relationships

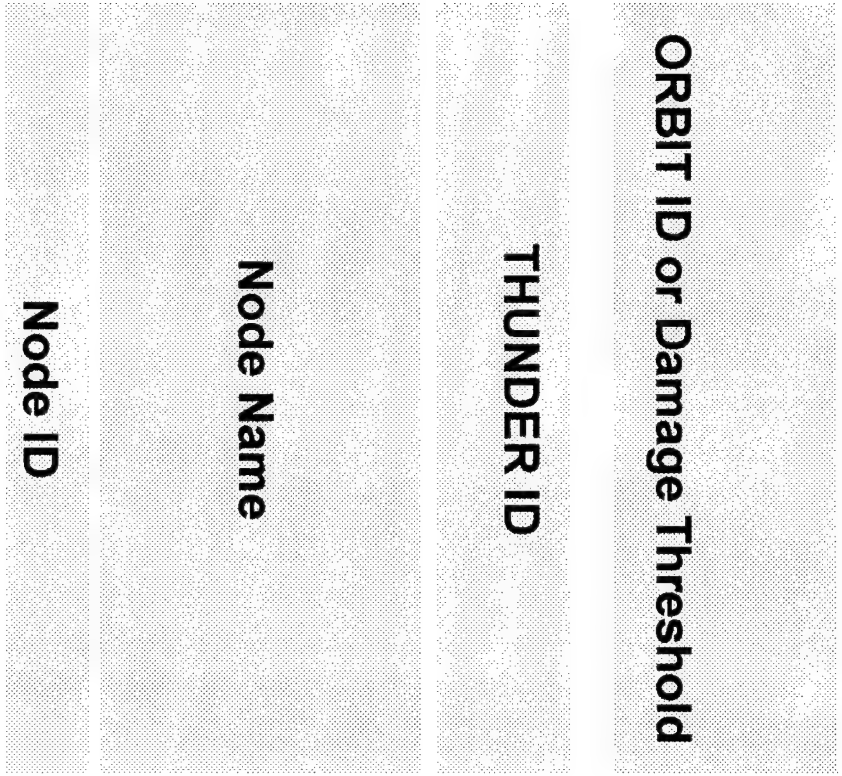
Latency Values

- **ACE-Link automatically generates Ordered Set of Loops with corresponding latency values**
  - Encodes performance of architecture
  - Encodes constraints driven by integration of operational and systems views in L-Edit



Network Entity Map

- **Maps L-Edit "Places" as nodes in the C2 Network to THUNDER discrete entities**
- **ACE-Link Automates format of file and and appropriate mapping of sub-model data structure to THUNDER**
  - Provides places to reference specific entities within sets
  - Provides places to define thresholds for damage that define "aliveness"



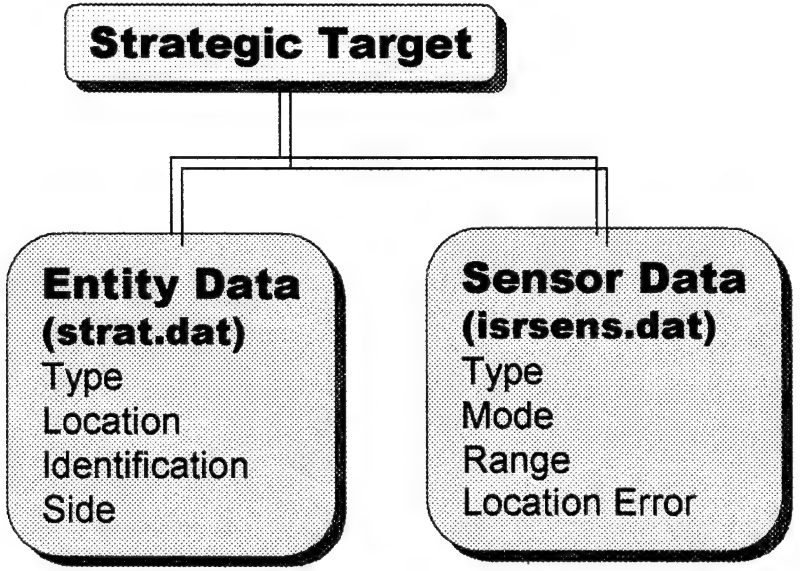
C2 Node Types

ACE-Link Creates Shell for Node Identification

- **Strategic Targets** **reflect ground based nodes in the network, both manned and autonomous**
  - Air Operations Center, EW Radar, MILSATCOM Downlink
  - Specific C3 vans, antennas, etc. must be moved to a "strategic target" to isolate the particular node
- **Satellites** **provide orbital nodes in sensor-shooter net**
  - Sensors like Space Based Infra-Red (SBIR) and Defense Support Program (DSP
  - MILSATCOM assets like UHF Follow-On (UFO)
- **Aircraft** **on Station carry sensors and act as communications relays for Line-Of-Site links**
  - JSTARS, AWACS

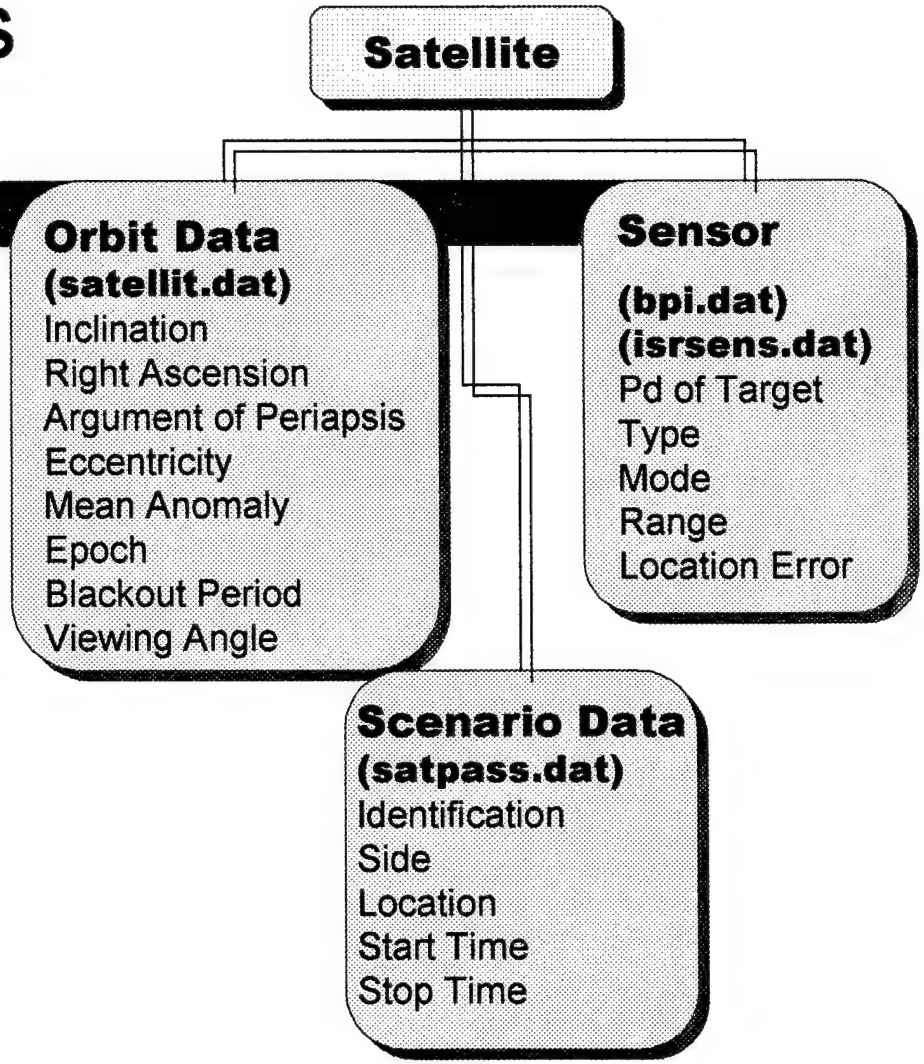
Ground C2 Nodes

- **THUNDER, as an Air-Combat simulation, typically aggregates ground targets into units**
- **Strategic Targets offer a formalism for explicit definition of individual Ground targets**
  - Geography
  - Order of Battle
  - "Stuff" they're made of
  - Are they detectable?



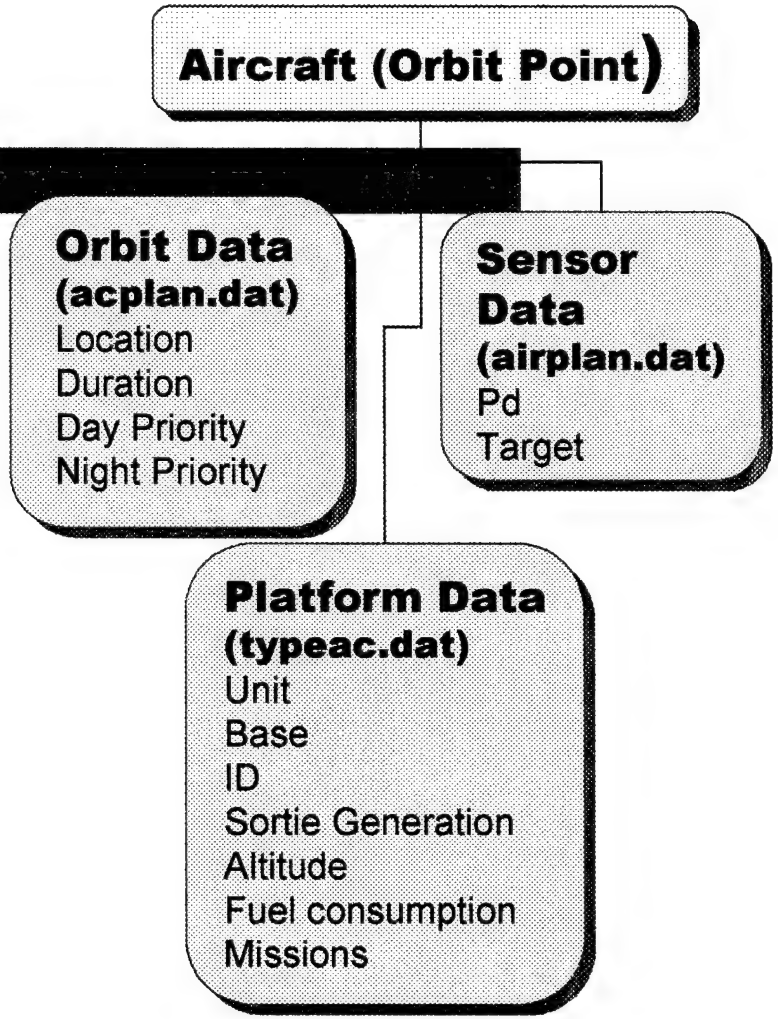
Satellite Nodes

- **Satellite nodes need unique descriptive data not encoded in L-Edits architecture information**
  - Orbital parameters
  - Time
  - Performance

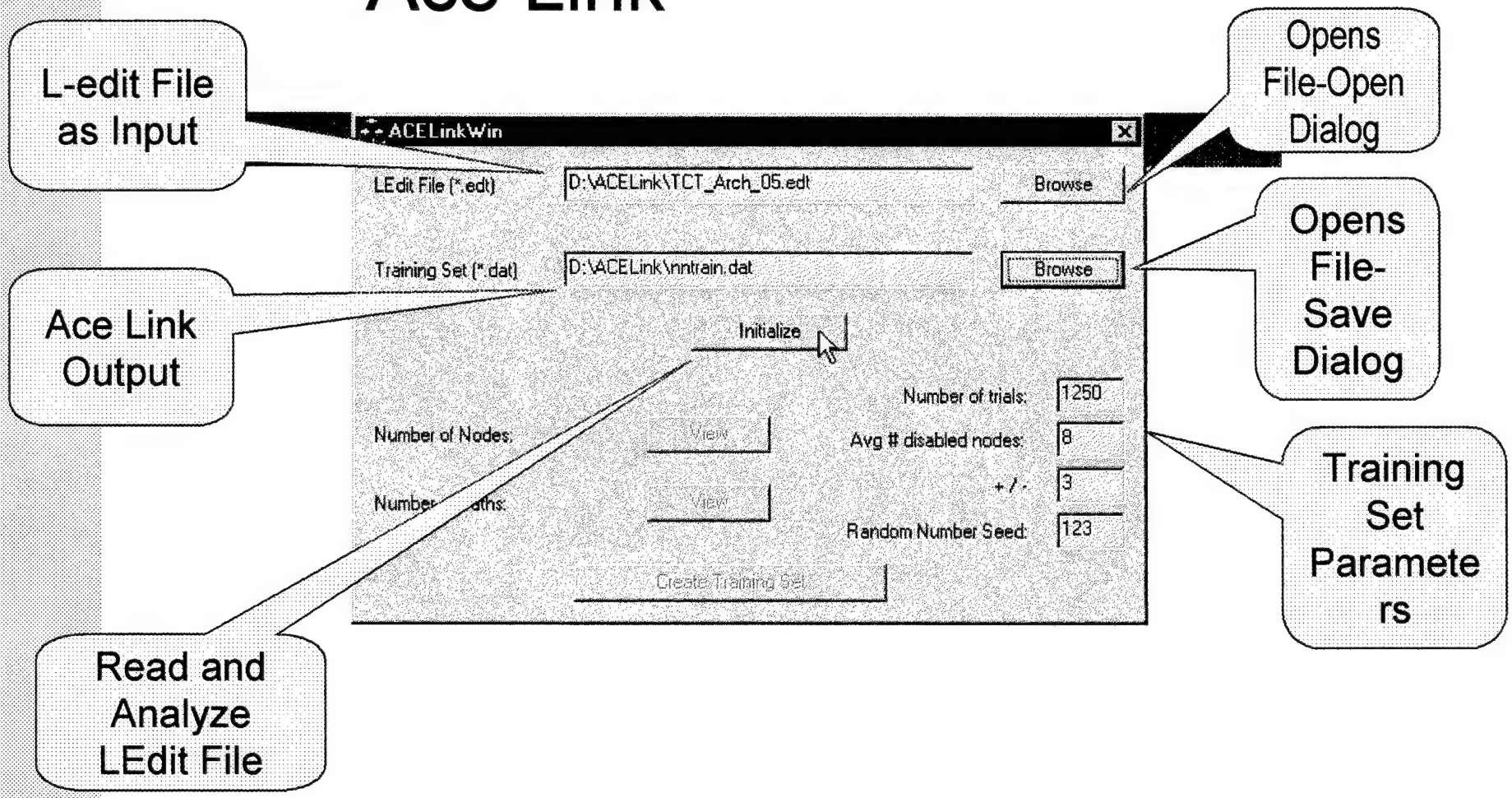


Aircraft Nodes

- **Aircraft Definition requires scenario-specific data that's mechanical, organizational and captures CONOPS**
  - Geographic
  - Order-Of-Battle
  - Performance
  - Mission

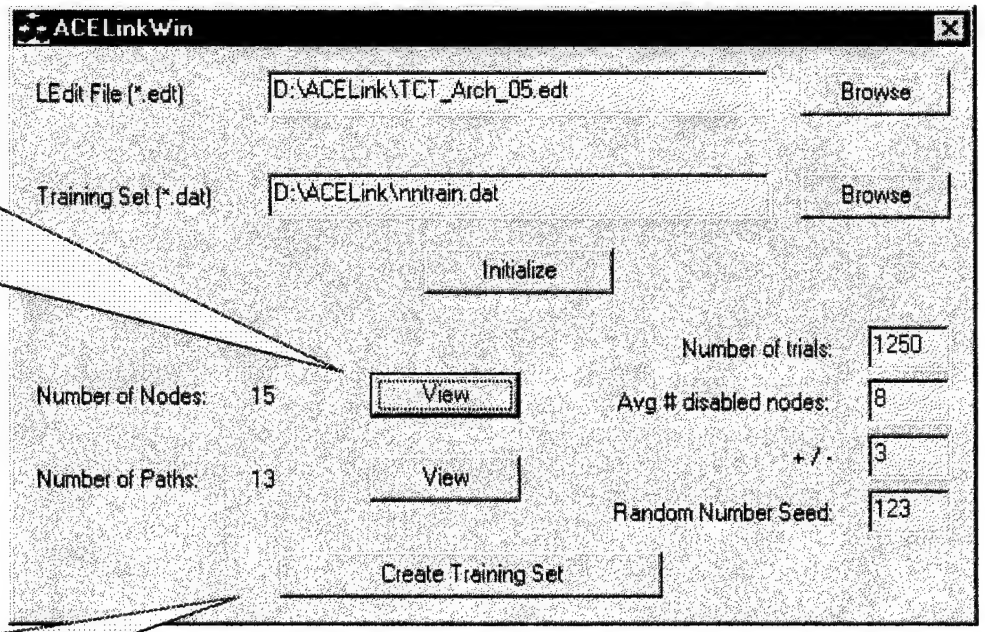


Ace Link


L-edit File as Input
Ace Link Output
Read and Analyze LEdit File
Opens File-Open Dialog
Opens File-Save Dialog
Training Set Parameters
ACELinkWin
LEdit File (*.edt)
D:\ACELink\TCT_Arch_05.edt
Browse
Training Set (*.dat)
D:\ACELink\nntrain.dat
Browse
Initialize
Number of trials:
1250
Number of Nodes:
View
Avg # disabled nodes:
8
+/-
3
View
Random Number Seed:
123
Create Training Set

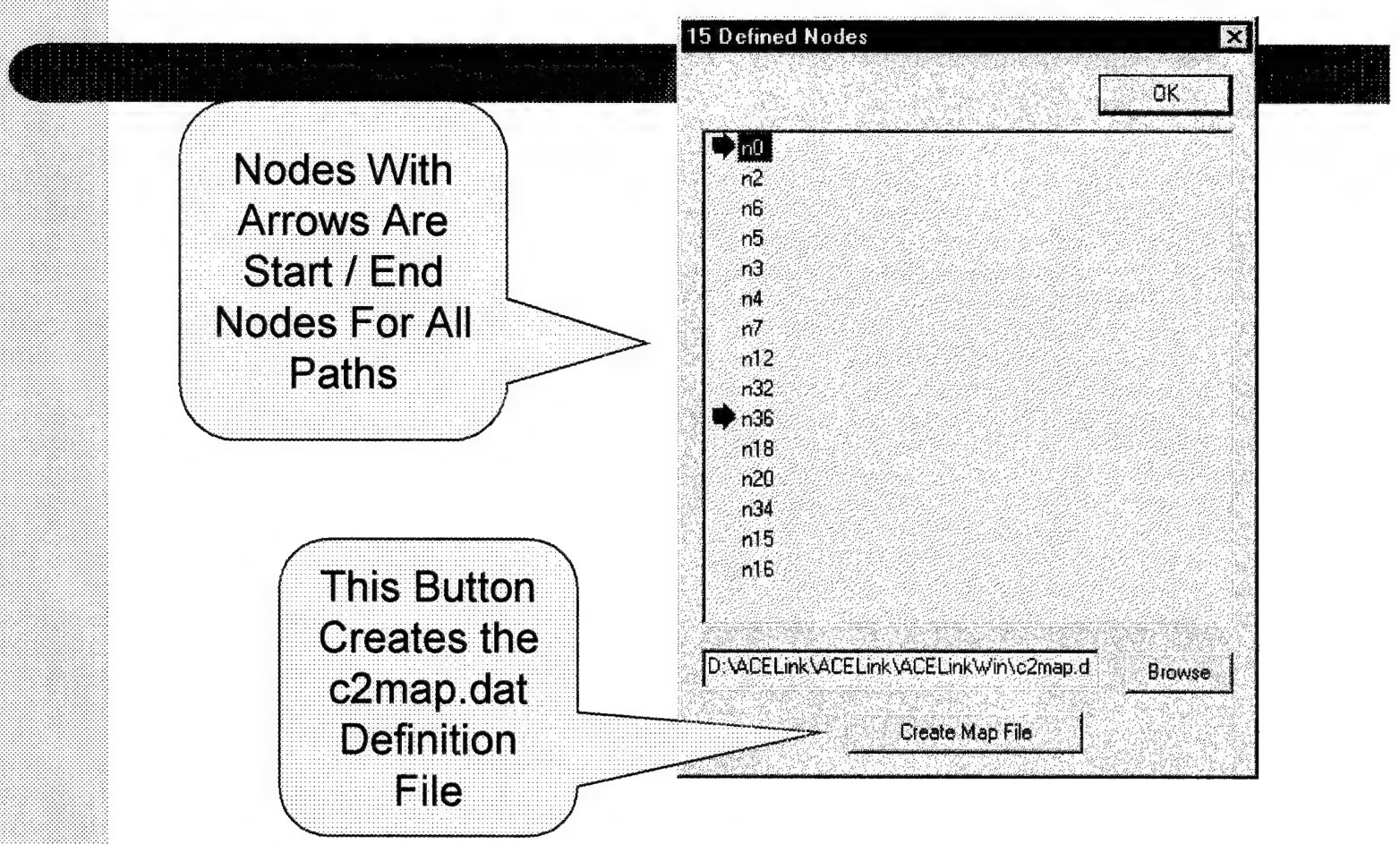
Ace Link

Once Initialized, Nodes and Paths Can be Viewed



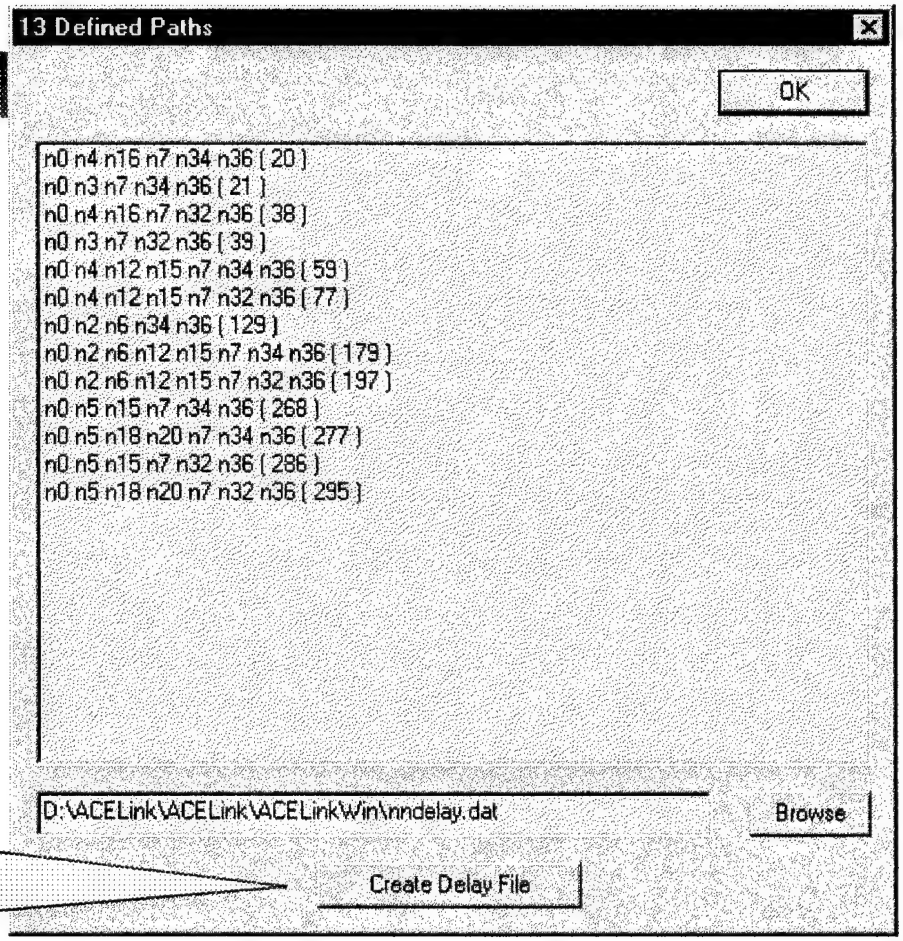
Create Neural Net Training Set

Ace Link—View Nodes



Ace Link—View Paths

Paths Are Sorted By Latency, Shown In Brackets



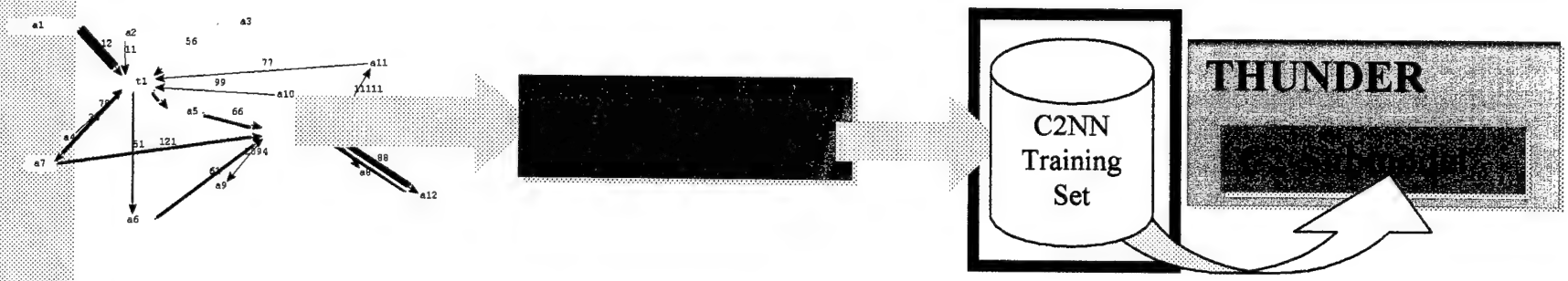
This Button Creates the nndelay.dat Description File

Training Set Algorithm

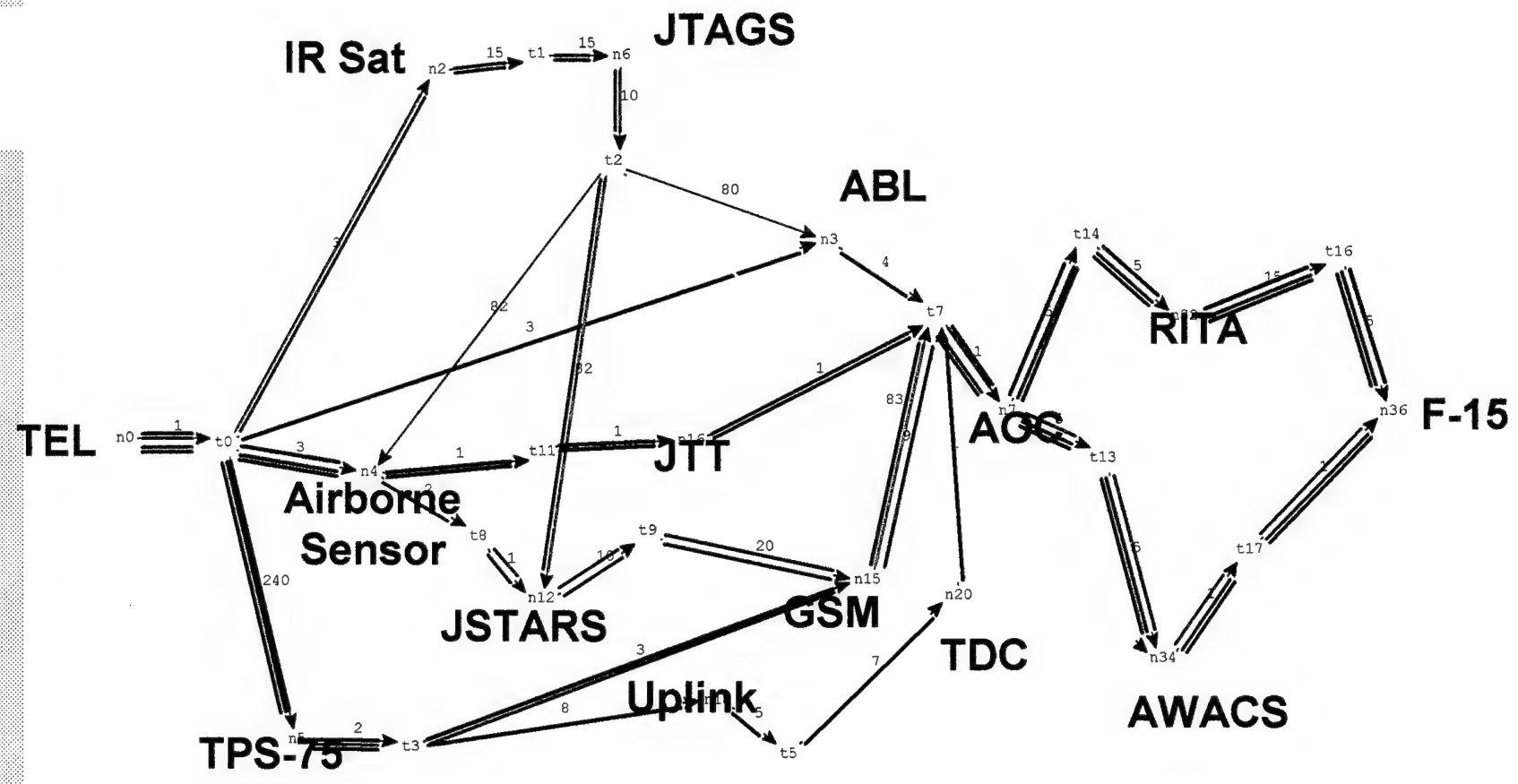
- **Create a Gaussian Distribution**
  - User Specified Mean Number of Disabled Nodes
  - User Specified Standard Deviation
  - User Specified Random Number Seed
- **For Each Trial (Total Number Specified by User)**
  - Pull The Number of Disabled Nodes From the Gaussian Distribution
  - Divide By Number of Nodes (Not Counting Start / Stop Nodes)
    - The Start / Stop Nodes Are NEVER Disabled
  - For Each Node (Excluding Start / Stop) Check a Uniform Random Number Between Zero and One
  - If Less Than Calculated Fraction Disabled, Disable For This Trial
  - Determine Best Enabled Path (Smallest Latency)

Demo Schematic

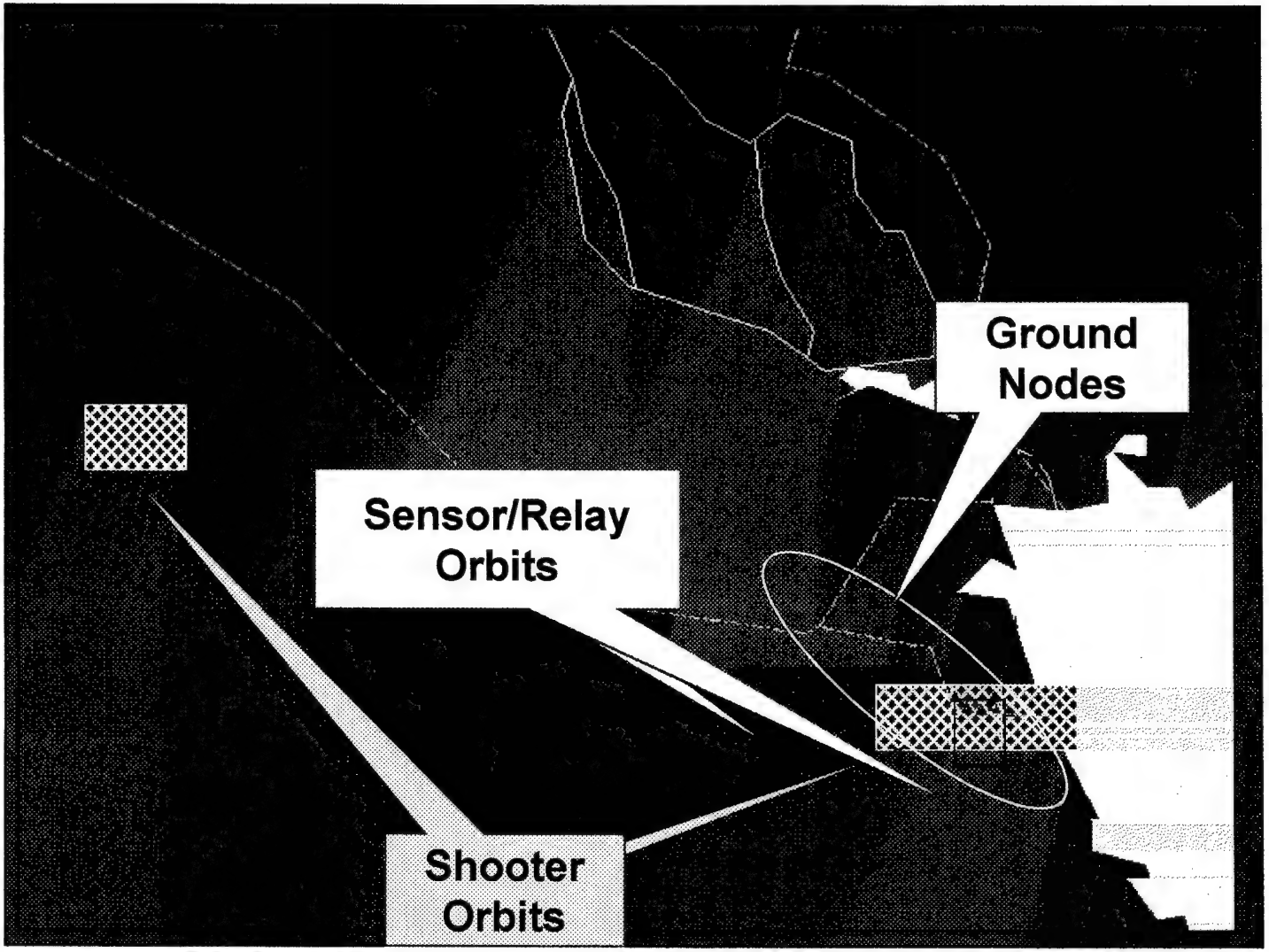
- We'll edit a baseline architecture for TCT mission
- We'll use ACE-Link to calculate performance and create data for simulation
- We'll train the C2-submodel
- We'll execute THUNDER with this C2 sub-model



TCT Architecture

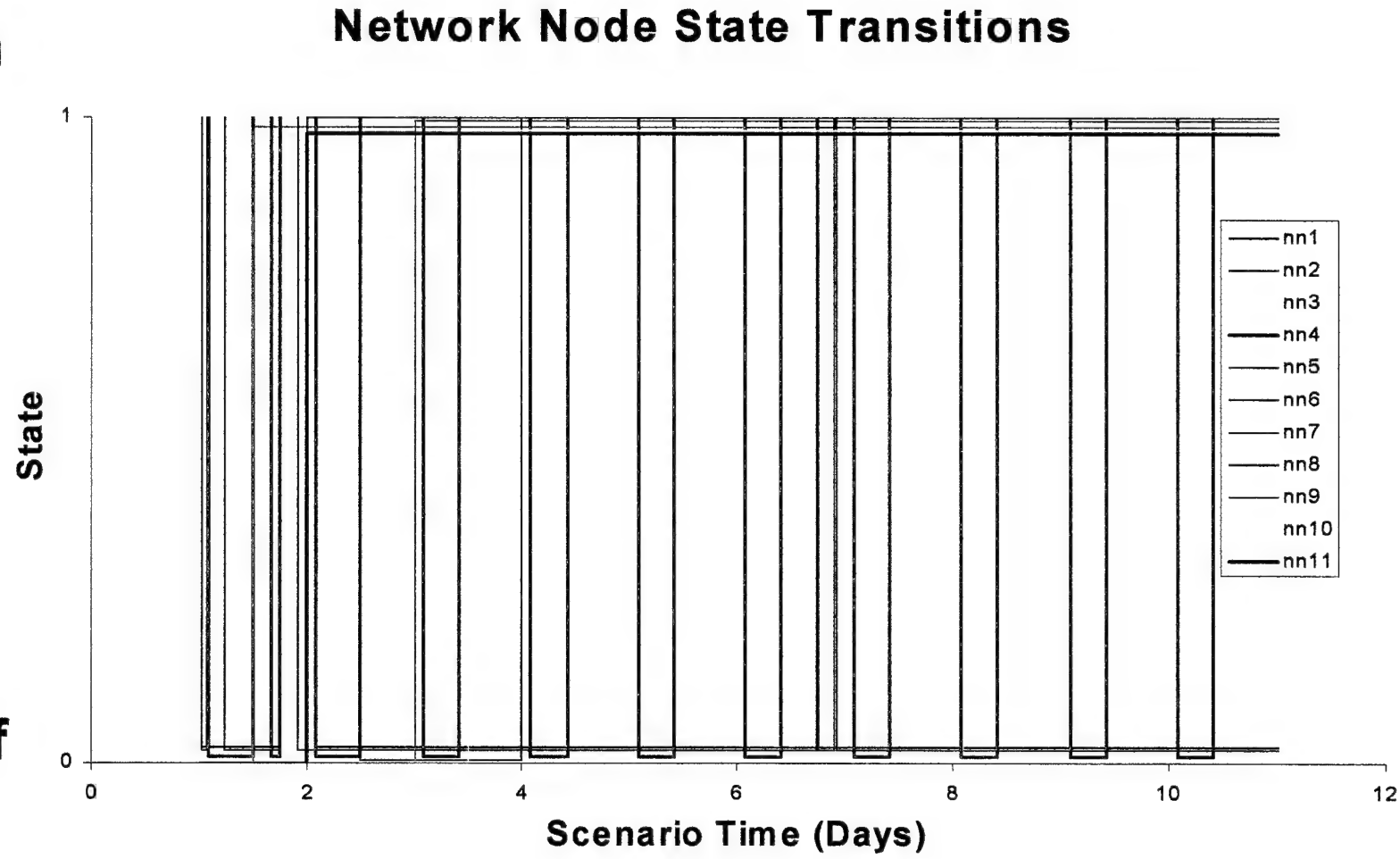


THUNDER MAP



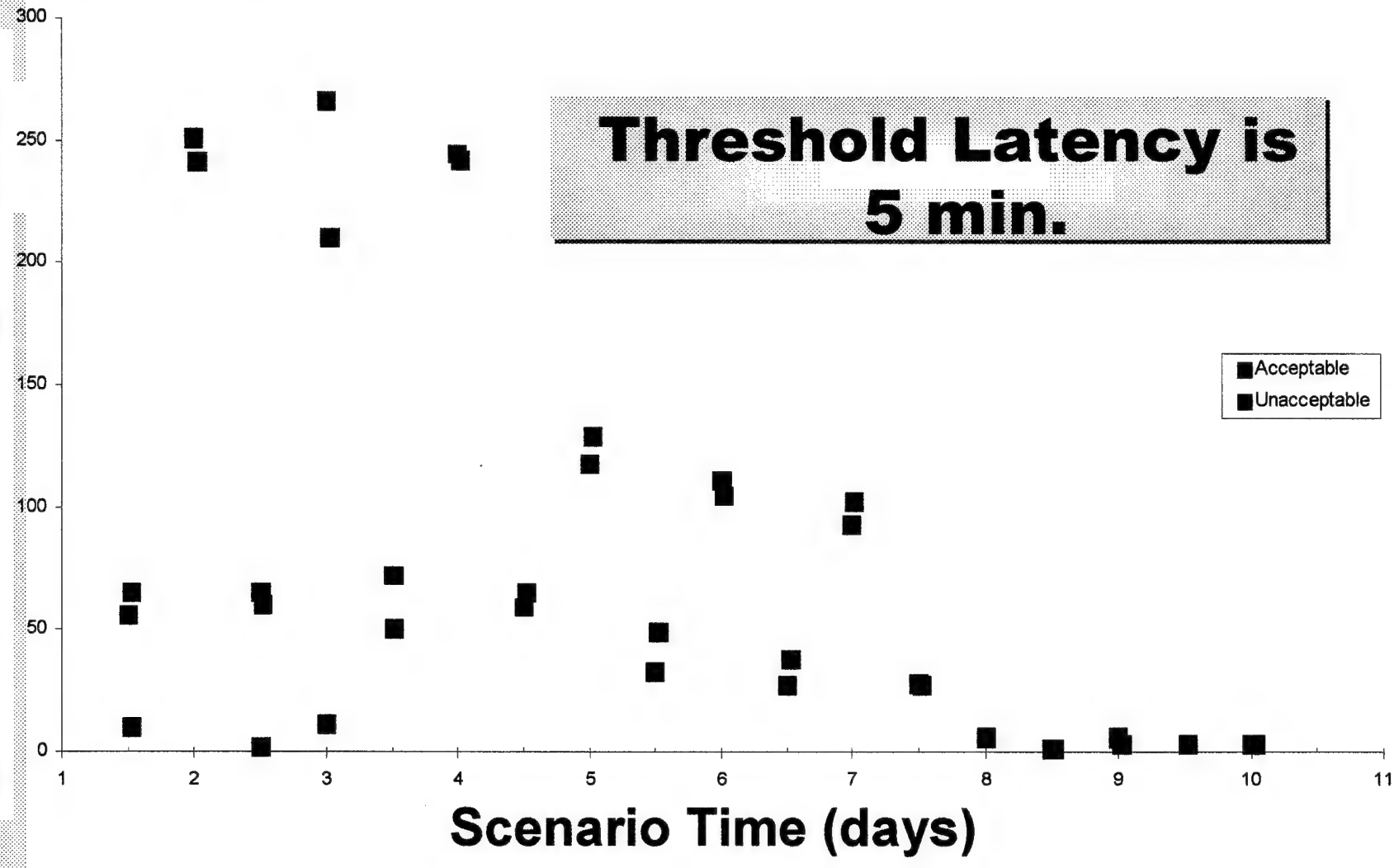
Changing Nodes

On



Off

Latency Distribution



Threshold Latency is 5 min.

What ACE-Link means to C2 Analysts

A Traceable Relationship Between Architecture and Combat Outcomes

- **Architecture alternatives captured in ESC/MITRE's tools can be evaluated in virtual combat experiments**
  - Exploit L-Edit formalism to parallel ESC and AC2ISRC efforts
- **Provides Military Worth insight in Time-Critical-Target domain**
  - High-priority C2 study area

Benefits

ACE-Link:

- Adds value to ESC's MRT tools, data and methods for evaluation of architecture alternatives
- Adds value to AF Studies and Analysis Agency's THUNDER simulation for supporting Air Staff decision-making
- Re-uses AFRL's model abstraction technology
- Takes advantage of Common Analytic Simulation Architecture (CASA) for extensibility

Extensions and Applications

The Proof-of-Concept is Successful, Providing *Relevant Capability* and an *Extensible Foundation*

- **Other mission areas**
  - Airborne Early Warning
  - Integrated Air Defense
- **STORM**
  - C2 Infrastructure with abstraction sub-model
- **AFRL/IF Collaborative Engineering Environment**
  - Requirements analysis
  - Analysis of Alternatives
- **Non-combat business interfaces both DoD and commercial**
  - Performance of warning dissemination networks for FEMA

Section II

ACE Link - An Approach to Integrating Command and Control Model Architectures (I)

Interim Report
June 14, 2000

Prepared by

Modasco Inc.
58 West Michigan Street
Orlando, Florida 32806
and
Emergent Information Technology - East
1700 Diagonal Road Suite 500
Alexandria, VA 22314

For

Department of the Air Force
Air Force Research Laboratory
Wright-Patterson Air Force Base, Ohio 45433

Under Contract
F33615-00-C-1669

Table of Contents

Executive Summary

The Modasco-EITE team investigated the current capability of LEdit, a graphical-design tool developed by the U.S. Air Force Electronic System's Center (ESC), to assist analysts in describing complex relationships. LEdit uses Colored Petri Nets (CPNs) as its design methodology and provides a mature set of features that can be applied to a wide variety of problems that occur in military and commercial applications. Two distinct commercial scenarios have been considered: (1) LEdit can be used as either a stand-alone tool, or can be integrated into other design and reporting tools, to graphically describe the complex relationships between analysis elements. (2) LEdit can be integrated into predictive models as a graphical design tool. In this case, LEdit would either be modified, or middleware developed, to provide input to the analysis engine of the model.

A number of deficiencies in the implementation of LEdit have been identified. These deficiencies are primarily related to the user interface, but some degradation in functionality has also been identified. Only one feature is judged to strongly limit the usefulness of LEdit as a graphical design tool. Therefore, LEdit, in its current state, is considered to have commercial applicability with only minor modifications.

The basic groundwork for integrating LEdit with the Air Force predictive modeling tool, THUNDER, is discussed. One Time Critical Targeting (TCT) function (U-2 target acquisition to air attack) has been described using LEdit. This function is shown to have four independent processes, which are represented by colored Petri Net loops.

1. Introduction

Modasco was provided an executable version of LEdit by the Government soon after the project start date of 8 May 2000. A detailed investigation was performed to determine the commercial viability of LEdit as a process-design tool. Two distinct commercialization scenarios were considered:

> LEdit has commercial appeal as a generalized process-design tool for business, industrial and government applications. Such functions as work flow, information processing, product distribution and software design can be graphically represented clearly and concisely using Colored Petri Nets (CPN). LEdit could be used as either a stand-alone design tool, or could be integrated into related programs such as PowerPoint, Word and Rational Rose.
>
> LEdit has broader applicability as the design element of a simulation and modeling tool. In this regard, Modasco has initiated the prototype development of ACE-Link, an integration of LEdit and THUNDER, an element of the Air Force analytic toolkit that measures force-level operational impact. LEdit could serve as a generic front-end to a variety of predictive tools used for the design of military command and control systems and commercial network-based communications systems.

2. LEdit As a Process-Design Tool

LEdit is a reasonably mature software tool that assists the user in describing generic processes. It is a Windows application written in Visual C++ consisting of approximately 150,000 lines of source code. Its development up until this point as been, according to its developers, somewhat unstructured, with no fully articulated functional and performance specification. As a result, some LEdit features are considerably less mature than others. This lack of uniformity can be annoying to the user and clearly detracts from its overall usefulness. However, it has two outstanding features that make it a significant and highly useful tool, even in its present state of development.

2.1 Process Highlighting

Most business and military functions have a complex structure in which events occur either randomly or based upon specified conditional logic. The resulting "flow chart" used to describe the function consists of branches occurring at each transition point, and, as the number of such transitions increases, the number of individual ways the function can occur increases. We refer to one specific set of conditions that results in the fulfillment of a specific function as a "process". In the pedagogical literature of computer science and operations research, the terms "thread" and "loop" are also used to describe this idea. LEdit allows the designer to enumerate the various processes of a function that can occur, highlight them graphically using color and isolate them

individually by hiding from view those that are not relevant to the specific focus of a discussion of the function. This provides the designer with a powerful tool for isolating a specific process from a group of many such processes during analysis and discussion of the function. Equally as important, a visualization of all such processes can indicate choke points through which many processes must pass for the function to occur. This provides the user with cues concerning the allocation of resources to ensure the function is performed efficiently. Figure 1 illustrates a simple function for which there are three processes.

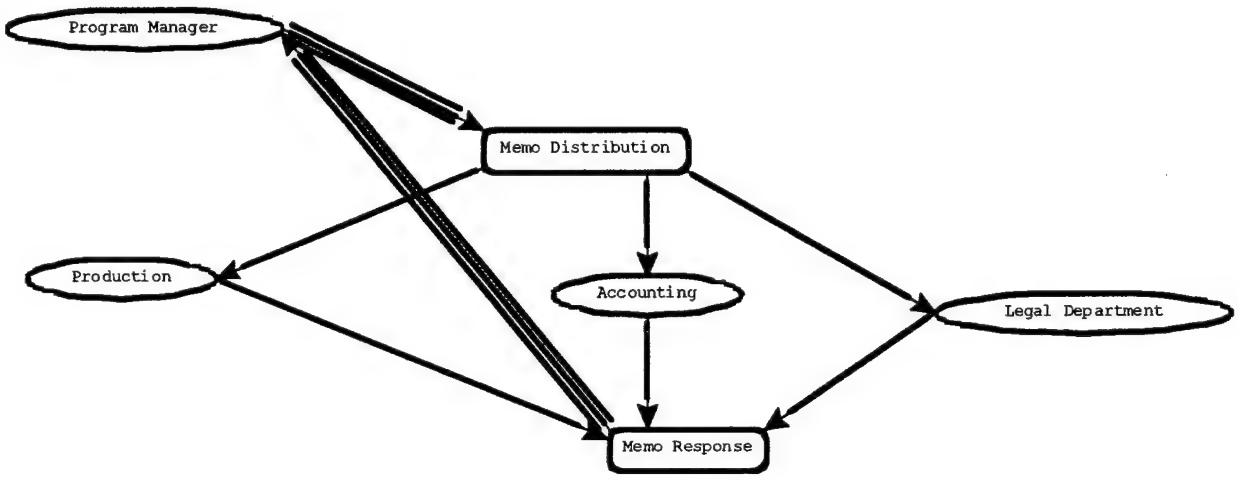


Figure 1: Simple Function With Three Processes
The three processes are colored red, blue and green to distinguish them. Each process represents one way that the function can be performed.

2.2 Extensibility

LEdit provides a powerful capability for designing and describing large, complex functions in a hierarchy of subfunction models. One can begin at the highest, most general level of description to model the basic structure of a function. Then, increasingly more detail can be added to the model by replacing each node with a more descriptive submodel describing its internal structure. Continuing in this way, the most specific details of a function can be embedded in the model. Each subfunction can then be treated graphically as a "black box", with specific input and output, but no internal structure is viewed in detail. By specifying which subfunctions are visible and which are hidden, the user can change the granularity of the function's visualization and thus apply focus to a specific part of it. Figure 2 illustrates the same function as described by Figure 1 with considerably more detail.

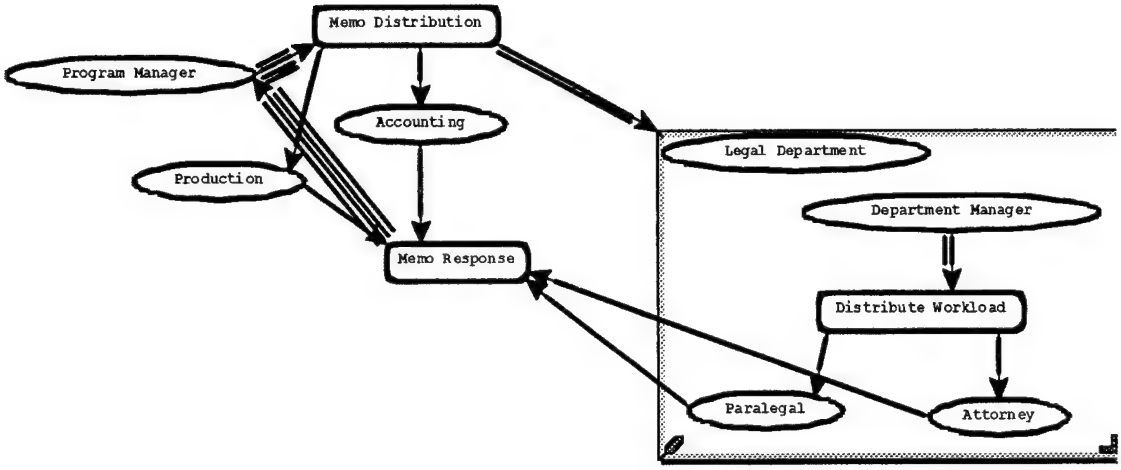


Figure 2: More Complex Function With Four Processes
This illustration extends the function described in Figure 1 by adding more detail to the right-node (labeled Legal Department), which now contains two subprocesses. These subprocesses are colored blue and orange. With this extension, the function now has four processes.

2.3 Deficiencies

While the two features of LEdit described in paragraphs 2.1 and 2.2 provide a powerful design capability to the user, there are a number of deficiencies in the way these features have been implemented that degrade its overall usefulness. These deficiencies are primarily related to the user interface, which is, at times, counter intuitive. Additionally, the method used to refresh the screen and to store designs in a graphical format are both less than optimal. A comparison has been made with two established software packages: Microsoft Word, whose user interface is the standard for Windows applications, and Rational Rose, which aids the user in producing Object Oriented analysis and design. Deficiencies are noted, described and ranked as a precursor to developing a work breakdown structure for development of LEdit as a commercial product. Each deficiency is evaluated on a criticality scale from 1 to 4. Table 1 contains a description of the features of each type of deficiency. Table 2 contains descriptions of the identified deficiencies.

Deficiency Level	Description
1	Mildly distracting, but does not degrade the overall performance of the software
2	Causes some confusion and increases the time required to perform some functions
3	Creates limitations that can only be overcome by complex workarounds.
4	Causes extreme confusion or is highly distracting; strongly limits the usefulness of the product

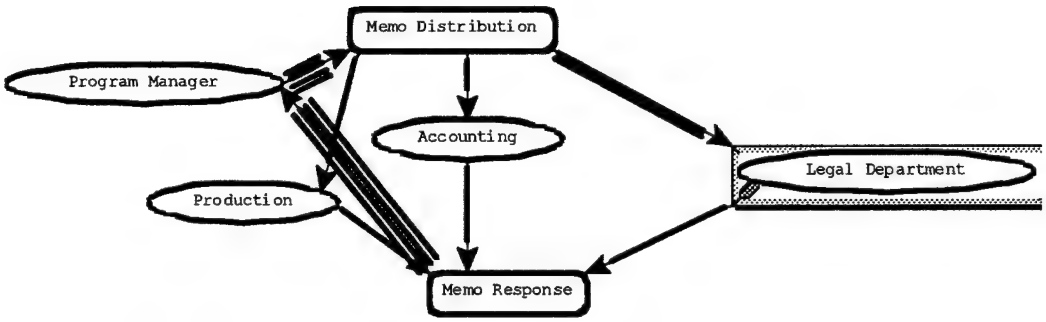


Figure: 3 Contracted Version of the Function of Figure 2.
The Place "Legal Department" has been contracted; however, one arc is drawn from the place to the Transition "Memo Response" and it is colored orange. In the expanded version of this Function, two arcs are drawn to this transition.

2.4 Summary

Fourteen (14) deficiencies have been identified and described in Table 2. Of these, only one is judged to be a Level 4 deficiency. As a result, LEdit is already sufficiently mature to provide useful modeling and design assistance. It is recommended that effort be expended in identifying potential markets for a design tool based upon the LEdit architecture.

Deficiency Number	Description	Recommended Corrective Action	Criticality
1	Icons (place, transition and arc) are diffuse and grainy.	Icons in Word and Rational Rose are clear and sharp.	1
2	The software palette contains ten (10) icons. It is not clear what the different icons represent or what they should be used for.	Use "Tool Tips" to provide a contextual description of each icon. Both Word and Rational Rose use "Tool Tips" extensively.	2
3	Icons are placed on the Design Window by selecting the icon (place cursor over icon and single left click the mouse), moving the mouse cursor to the desired location in the Design Window and double left clicking the mouse.	Both Word and Rational Rose place the icon in the Design Window with a single mouse click. Since LEdit uses a single mouse click to select an icon, it should also use a single mouse click to place it in the Design Window.	2
4	Once an icon has been placed in the Design Window, the user double left clicks the mouse to display Dialog Boxes for expanding the description of the icon.	In both Word and Rational Rose, this function is performed by a single right click of the mouse. This is a standard that somewhat distinguishes Windows applications from Macintosh applications.	2
5	It is not possible to draw two distinct arrows between a place and a transition. LEdit draws a single line with "arrow heads" on each end to depict data flow in both directions. However, it is not possible to add two distinct textual descriptions to the "double arrow".	Allow two distinct arcs to be drawn between a place and a transition.	4
6.	All arcs are drawn as straight lines. Thus, it is common for an arc to be drawn over other icons. This makes the graphical design more complex to visually interpret. Figure 1 of this report clearly shows this effect: all three arcs drawn from the bottom-most place to the top-most place pass through the middle of the graphic.	Allow arcs to be drawn as curves. LEdit will initially draw each arc as a straight line. The user may select the line at any point along its extension and drag that point to any specified point in the Display Window. The intermediate form of the arc will be two straight lines that join at the specified point. LEdit will then draw a circular arc passing through the three points (place, transition and specified point) that define its perimeter	1

7.	All icons (places, transitions and arcs) are re-drawn every time a change is made to the graphical display.	Only affected icons should be redrawn. Refreshing the entire graphic is visually annoying and time consuming. It causes the user to loose focus on the change just made while other parts of the graphic are redrawn.	2
8.	A graphical design created in LEdit is stored in an "Enhanced Meta File" (.emf) format.	This format is not entirely compatible with some newer versions of Microsoft Word. Better choices would be the .tif, .gif, .jpg formats.	3
9.	There are no visual cues to aid the user in selecting and copying/cutting sections of LEdit designs.	There are two ways of providing such feedback to the user: (1) Display a dashed box around the selected area (Rational Rose) (2) Highlight sections of the design that are selected by adding hash marks along the periphery of the icon (Word)	3
10.	The interface for displaying/hiding individual processes (loops) is cumbersome.	Provide a Tool Bar with one Radio Button for each process. The user can then select the processes (loops) that should be displayed and hidden. Each Radio Buttons should have the same colors as processes (loops)	2
11.	The user can toggle any place (expand or contract the display of the details of the place) by a single left mouse click on the tab in the lower-left corner of the place display box. However, the region of the tab that will capture the mouse click is small and it often takes the user several attempts to toggle it.	Expand the region of the tab that can be used to toggle the place.	2
13.	Contracting a place with two or more arcs emanating from it produces a result that is not consistent with the intent of the user. For example, when the place labeled "Legal Department" in Figure 2 is contracted, the result is Figure 3, with a single arc colored orange.	The single arc emanating from a contracted place should have all of the colors of the processes (loops) that include this place.	3
14.	Contextual Help is not provided. Instead, a version history of LEdit is provided.	Version history should be replaced by Help About.. messages.	3

3. LEdit As an Interface to Predictive Modeling and Analysis Tools

3.1 Background

The current state of combat campaign analysis tools does not allow resolution of the impact of changes in Command and Control (C2) architecture. **Joint Vision 2010** points to a future for combat operations enabled through "Information Superiority", defined *as "the capability to collect, process, and disseminate an uninterrupted flow of information while exploiting or denying an adversary's ability to do the same."* (Office of The Chairman, The Joint Chiefs Of Staff, Washington, D.C. 1997.) Clearly, this reliance on dominant battlefield information places a premium on the underlying C2 infrastructure. With the JV2010 implementation envisioned in Network Centric Warfare we find *"...information superiority is a comparative or relative concept. Furthermore, its value is clearly derived from the military outcomes it can enable."* (D.S.Alberts, J. J. Garstka. and F. P. Stein. **Network Centric Warfare: Developing and Leveraging Information Superiority**. DoD C4ISR Cooperative Research Program Publication Series, Washington DC, 1999). These trends point to a growing need to evaluate critical C2 alternatives in terms of military worth.

THUNDER, the campaign simulation in the Air Force Suite Of Models for Analysis (AFSOM-A), does offer some sensitivity to aggregate measures of performance of the C2 domain as a whole, generally described by processing delays. These delays are characterized by distributions and user-defined rule-sets in the version released in May 2000 (V6.7). THUNDER provides no inherent traceability between these descriptions of C2 performance and the structure and state of the underlying architecture. With the emergence of information-centered warfare concepts, it's precisely this analytic thread connecting architecture to force-level combat effectiveness where analytic capability is needed. This effort addresses near-term solution in THUNDER and lays the foundation for improved C2 analytic capability in STORM.

3.2 Overview of Improved Methodology

Our approach provides more direct traceability for C2 behavior and more dynamic performance over the course of a campaign spanning tens of days of combat. The ACE-Link effort creates traceable connections between detailed models of C2 architectures and their effect at the campaign level. We will use ACE-Link to connect architectures described as CPNs to THUNDER C2 performance data through a Neural Network sub-model. The CPN will be exploited to develop data relating network state to performance. Network state will reflect the nodes and links that constitute the C2 architecture. Performance, in our phase 1 work, will be exclusively measured by tasking delay. The Neural Network "learns" the relationships between state and performance, recognizing key nodes and patterns of nodes that correspond to mode-changes in network performance. That is, it knows what stable, shortest path

is available given an architecture where some nodes may not be enabled. Early in a campaign, some nodes may not be deployed into the fight as they flow into theater from home station. Throughout the campaign, some nodes may be degraded or destroyed due to efforts of the opposing combatants. This method maps the evolution in the network topology throughout the campaign to changes in its ability to move information. This connection between action in the campaign and performance allows us to observe dynamic changes in the architecture's performance. That is, the performance changes through simulation time as the architecture itself changes with entities entering and leaving the network. The existing tool-set to derive this state-performance data is a rudimentary, proof-of-concept prototype. Integration with more robust network analysis tools extends the rough implementation to a much more credible and useful suite.

4. Operational Domain

The approach we're researching extends exploration supported by the Air Force Research Laboratory (AFRL) improving the ease of generating detailed data, developing more credible data and integrating with the Electronic Systems Center's developing tools and methods for C2 analysis. The CPN tools offer a powerful means to develop accurate, comprehensive models of C2 architectures. Even this improvement in the ability to elicit and present information gives us better capability in developing representations of C2 architectures for THUNDER and STORM. The simulation capability offered by the CPN methodology is an improvement to the simple algorithms used to prototype the C2 sub-model for THUNDER. The CPN-based models enable enhanced resolution of queuing and other factors impacting performance with the promise of integrating multiple C2 threads. Finally, the method is compatible with a growing foundation of C2 models developed with the operational experts as validated representations of C2 architectures. ACE-Link is the natural synthesis of leading-edge capabilities in C2 analysis.

4.1 Time-Critical-Targeting (TCT)

Our application will cover the tactical area of Time-Critical-Targeting (TCT). This is a natural domain for the method as it is sensitive to our principal performance measure, delay. The architecture we will use is derived from :

- **"Combat Air Forces Concept of Operations for Command and Control Against Time Critical Targets"**, issued out of HQ ACC/DRAW, 8 July 1997
- Joint Pub 3-01.5 "**Doctrine for Joint Theater Missile Defense**", (1996)
- AFTTP(I) 3-2.24 **"Multiservice Procedures for Joint Theater Missile Target Development"** (1999)

Our choice of this operational domain is motivated by the availability of data, accessibility in THUNDER/STORM and combat sensitivity. The TCT area is very stressful to C2 architectures and of high interest both operationally and in terms of systems development.

The architecture defines a sensor-decision-maker-shooter information flow. This mission illustrates the sensor-grid, decision applications and actor-grid interactions that form the foundation of the Network-Centric Warfare approach. The architecture is stimulated by detection of a Theater Ballistic Missile Launcher through detection of the launcher itself or detection of an associated launch event. Sensors route detection information along with sufficient information to identify the launcher to a decision maker. An intermediate network of communications and intelligence processing nodes provides alternate routes. Note that in today's operational architecture, procedures centralize the decision authority at the Air Operations Center. Beyond the decision-maker, information is routed to the selected platform to prosecute the launcher. Again, intermediate processing and communications nodes provide alternative routes.

4.2 ACE-LInk Definition

ACE-Link provides a capability to exploit ESC's work in modeling C2 architectures in military worth analysis. Ongoing work consists of documenting architectures for the Air Operations Center, Integrated Broadcast Service and other important operational implementations as CPN models. ACE-Link connects these models as they become validated to credible combat analysis capability at the campaign level. In parallel, we will finalize integration of the Neural Network (NN) sub-model in THUNDER's logic for prosecution of Theater Ballistic Missile (TBM) launchers. The ACE-Link capability automates some of the steps in developing the data to support the sub-model. The early prototype network editor only presented the architecture as a weighted directed graph. The graphical editor for the CPN architecture models is far superior in quality and user friendliness. The ability to develop meaningful icons and enforcement of bipartite relationships adds both intuitive presentation and rigor. This alone would facilitate the development of C2 architecture models. These models could feed the prototype shortest-path tool to provide training sets for the NN sub-model. As an evolution, the CPN simulation provides a more rich characterization of network performance. Integration of the executable component promises improved training sets for architecture performance characterization. The demonstration of capability builds near term analytic possibilities for exploring alternative architectures and procedures for this mission area. In addition, the lessons learned along with the algorithms and code developed here provide a foundation for improved C2 representation in the next-generation campaign simulation, STORM.

4.3 U2 Acquisition to Attack

One aspect of battlefield management against a TBM threat begins with target acquisition by a U-2's radar sensors, which provide both Moving Target Information and images, and ends with a terminal air defense mission. Figure 4 illustrates the details this function using CPNs generated by LEdit. There are four processes that can occur:

Process 1 (Red) - The target is acquired by a U-2; the CARS ground station is within Line-Of-Site (LOS) of the U-2 and processes data from the U-2 sensor suite; Centralized C2 decision authority is exerted by Decision Maker; AWACS conducts battle management in attack operations mission and controls air assets to target.

Process 2 (Green) - The target is acquired by a U-2; the CARS ground station is within Line-Of-Site (LOS) of the U-2 and processes data from the U-2 sensor suite; Centralized C2 decision authority is exerted by Decision Maker; a RITA communications van provides formatted imagery to air asset cockpits to assist in targeting.

Process 3 (Blue) - The target is acquired by a U-2; the CARS ground station is beyond Line-Of-Site (LOS) of the U-2 so that a MOBSTER ground relay station uplinks the U-2 sensor data to a communications satellite (SATCOM), which downlinks it to CARS; CARS transmits this data back to CONUS over SATCOM using Theater Deployable Communications (TDC). Centralized C2 decision authority is exerted by Decision Maker; AWACS conducts battle management in attack operations mission and controls air assets to target.

Process 4 (Orange) - The target is acquired by a U-2; the CARS ground station is beyond Line-Of-Site (LOS) of the U-2 so that a MOBSTER ground relay station uplinks the U-2 sensor data to a communications satellite (SATCOM), which downlinks it to CARS; CARS transmits this data back to CONUS over SATCOM using Theater Deployable Communications (TDC). Centralized C2 decision authority is exerted by Decision Maker; a RITA communications van provides formatted imagery to air asset cockpits to assist in targeting.

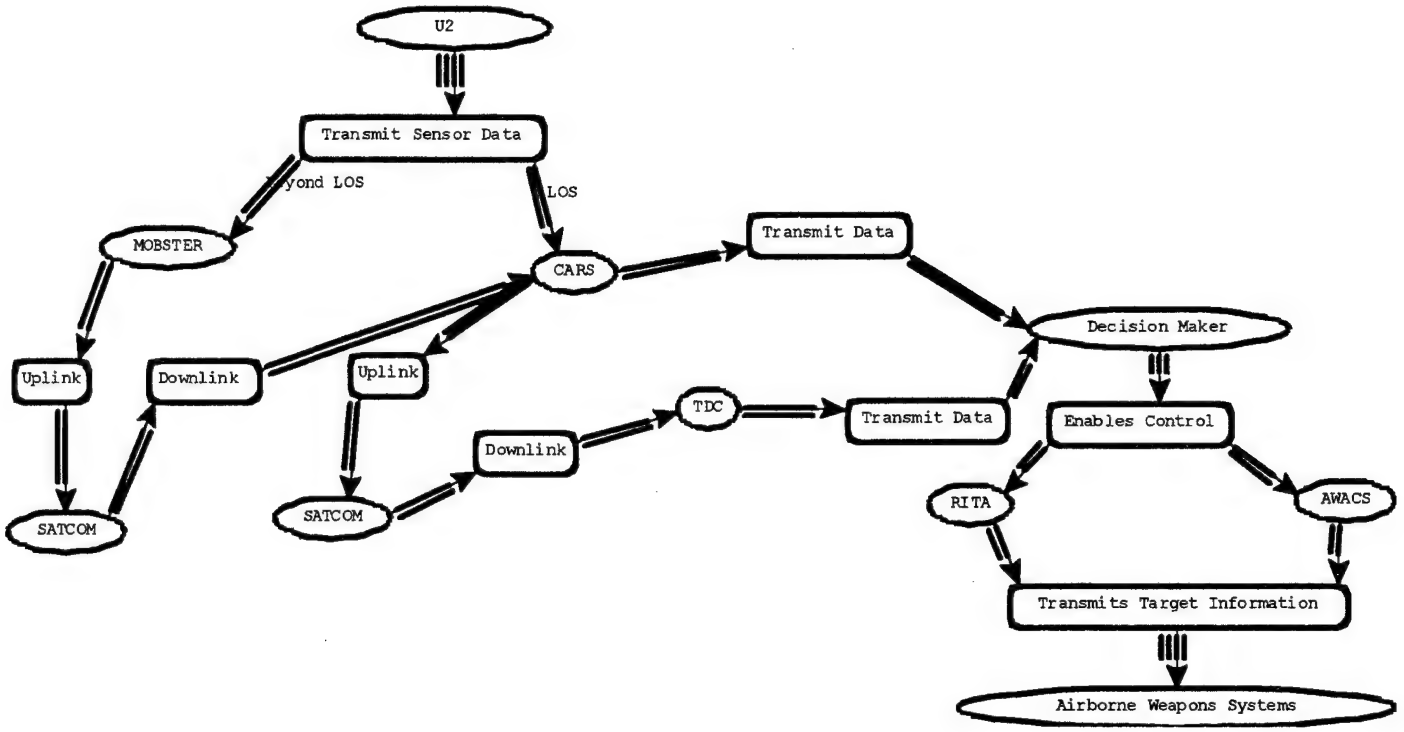


Figure 4: U2 Target Acquisition to Attack

The four independent processes that can occur are shown using LEdit to construct a CPN model of this function.

Deficiency Level	Description
1	Mildly distracting, but does not degrade the overall performance of the software
2	Causes some confusion and increases the time required to perform some functions
3	Creates limitations that can only be overcome by complex workarounds.
4	Causes extreme confusion or is highly distracting; strongly limits the usefulness of the product

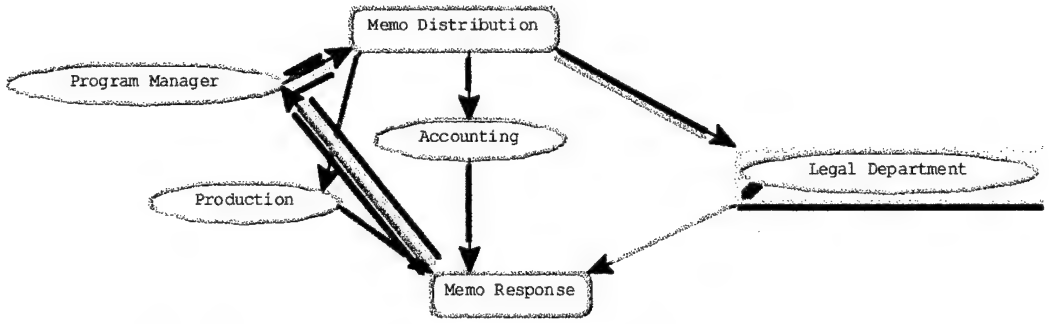


Figure: 3 Contracted Version of the Function of Figure 2.
The Place "Legal Department" has been contracted; however, one arc is drawn from the place to the Transition "Memo Response" and it is colored orange. In the expanded version of this Function, two arcs are drawn to this transition.

2.4 Summary

Fourteen (14) deficiencies have been identified and described in Table 2. Of these, only one is judged to be a Level 4 deficiency. As a result, LEdit is already sufficiently mature to provide useful modeling and design assistance. It is recommended that effort be expended in identifying potential markets for a design tool based upon the LEdit architecture.

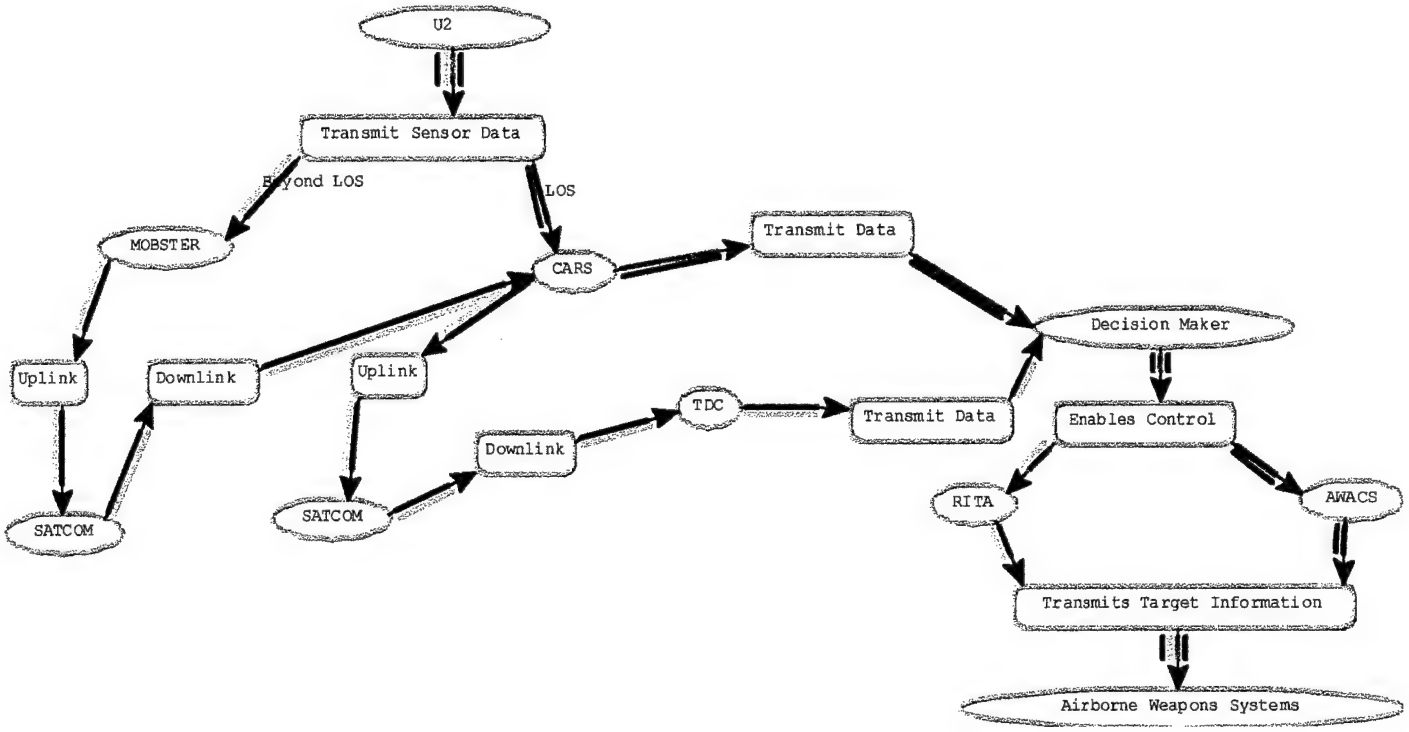


Figure 4: U2 Target Acquisition to Attack

The four independent processes that can occur are shown using LEdit to construct a CPN model of this function.

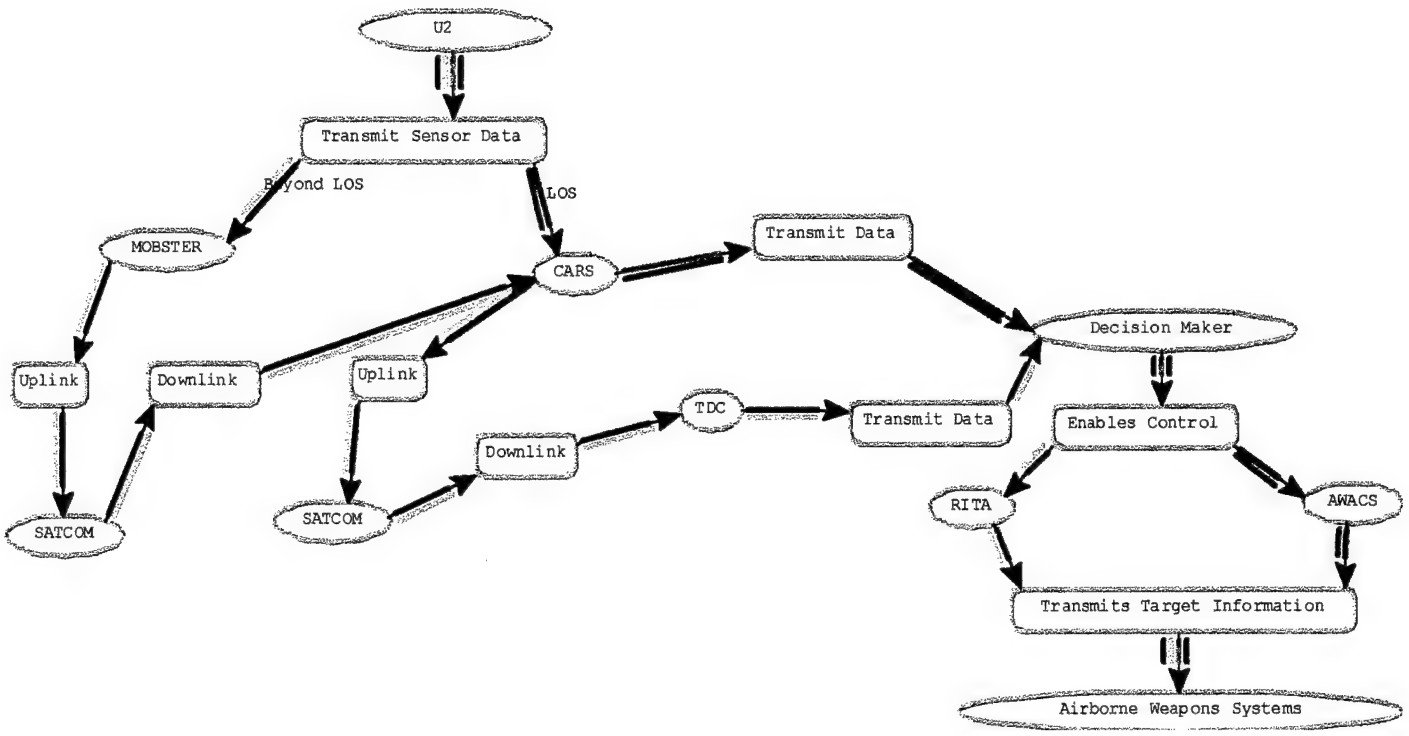


Figure 4: U2 Target Acquisition to Attack

The four independent processes that can occur are shown using LEdit to construct a CPN model of this function.

Section III

ACE Link - An Approach to Integrating Command and Control Model Architectures (II)

Interim Report
July 15, 2000

Prepared by

Modasco Inc.
58 West Michigan Street
Orlando, Florida 32806
and
Emergent Information Technology - East
1700 Diagonal Road Suite 500
Alexandria, VA 22314

For

Department of the Air Force
Air Force Research Laboratory
Wright-Patterson Air Force Base, Ohio 45433

Under Contract
F33615-00-C-1669

Table of Contents

Executive Summary

An investigation was conducted to determine the best method for developing a computational interface between LEdit and THUNDER. LEdit is a graphical tool that can be used to design communications networks. THUNDER is an Air Force predictive model that measures force-level operational impact. A method was developed, and described in detail in this Technical Report, for extending the usefulness of LEdit by embedding Time Critical Targeting (TCT) data in its network designs. The Figure of Merit for evaluating competing communications paths is the minimization of the time delay between transmission and reception of targeting information. It was found that

1. A straightforward computation of the optimal communications path by evaluating all possible paths between two communications nodes would be computationally restrictive.

2. LEdit provides a direct way of specifying the communications paths that can actually occur between two communications nodes, thus greatly reducing the number of such paths that must be considered in computing the optimal path. LEdit refers to a sequence of transitions between communications nodes as a "loop".

3. Using the methodology described in this Technical Report, sufficient information can be extracted for LEdit's output file to determine the optimal communications path and provide the necessary input to THUNDER.

It is recommended that middleware be developed to extract communication path information from LEdit , calculate the optimal communications path based upon a list of currently operational communication nodes provided to the middleware by THUNDER and provide the necessary input to THUNDER so that it can model Time Critical Targeting operational effects. The proposed middleware would pass back to THUNDER a description of the optimal communications path consisting of (a) the time delay for the communications path (2) a sequenced list of communications nodes that comprise the communications path.

A detailed Interface Design Specification (IDS) for the middleware is also presented. Tables 1, 2 and 3 describe the processing steps that must be implemented to extract the necessary path and loop information from LEdit output files. This IDS will be used in the next part of this investigation to develop prototype middleware to implement the proposed methodology.

1. Introduction

ACE Link is a fusion of two technologies: (1) Graphical design using Colored Petri Nets (CPNs) and (2) a predictive model that measures force-level operational impact. The goal of this investigation is to develop a straightforward way of integrating LEdit and THUNDER, two examples of these technologies. Since both LEdit and THUNDER are likely to be replaced by next-generation models in the future, a secondary goal is to formulate a generalized strategy for integrating design and modeling tools that can be applied to both current architectures and those that will be employed in the future. Therefore, the integration of LEdit and THUNDER will be achieved with minimal modifications to either program. This technical report describes the results of investigations performed by the Modasco-Emergent team and serves as an Interface Design Document (IDD) for realizing a specific example of ACE Link.

1.1 Background

Technical Report I (June 8, 2000) described an application of LEdit to Time Critical Targeting (TCT). Figure 1 illustrates four independent communications paths that may occur when a U-2 detects a Theater Ballistic Missile (TBM) launcher. There are ten (10) communications nodes, each depicted by an ellipse and representing either a receiver or transmitter of information. There are twelve (12) communications paths between nodes, each represented by a directed arrow. A communications path indicates that information can be transmitted between the two nodes only in the direction of the arrow. While bi-directional information flow is allowed (and is represented by a bi-directional arrow), it does not occur in this example. These transitions provide four (4) independent communications paths between the U2 and the airborne weapons system that will attack the missile site. These statistics are specific to this example and do not apply to a generalized communications network. A robust battlefield simulation should consider all four transmission paths and determine which path provides the most efficient means of directing an attack. It should also consider that, in general, one or more of these paths may be deteriorated or unavailable. Currently, this capability does not reside in either LEdit or THUNDER. Using LEdit, an analyst can design a communications network graphically as indicated by Figure 1. THUNDER will provide a battlefield assessment given a specific communications path. However, a comprehensive model that automatically reacts to different communication-network capabilities does not currently exist.

1.2 ACE Link Capabilities

The restrictions and limitations inherent in THUNDER diminish its usefulness as a predictive model, since the analyst must pre-process information concerning communication paths to determine which is the most efficient of those currently available. The development of an integrated design and modeling tool will require the following items:

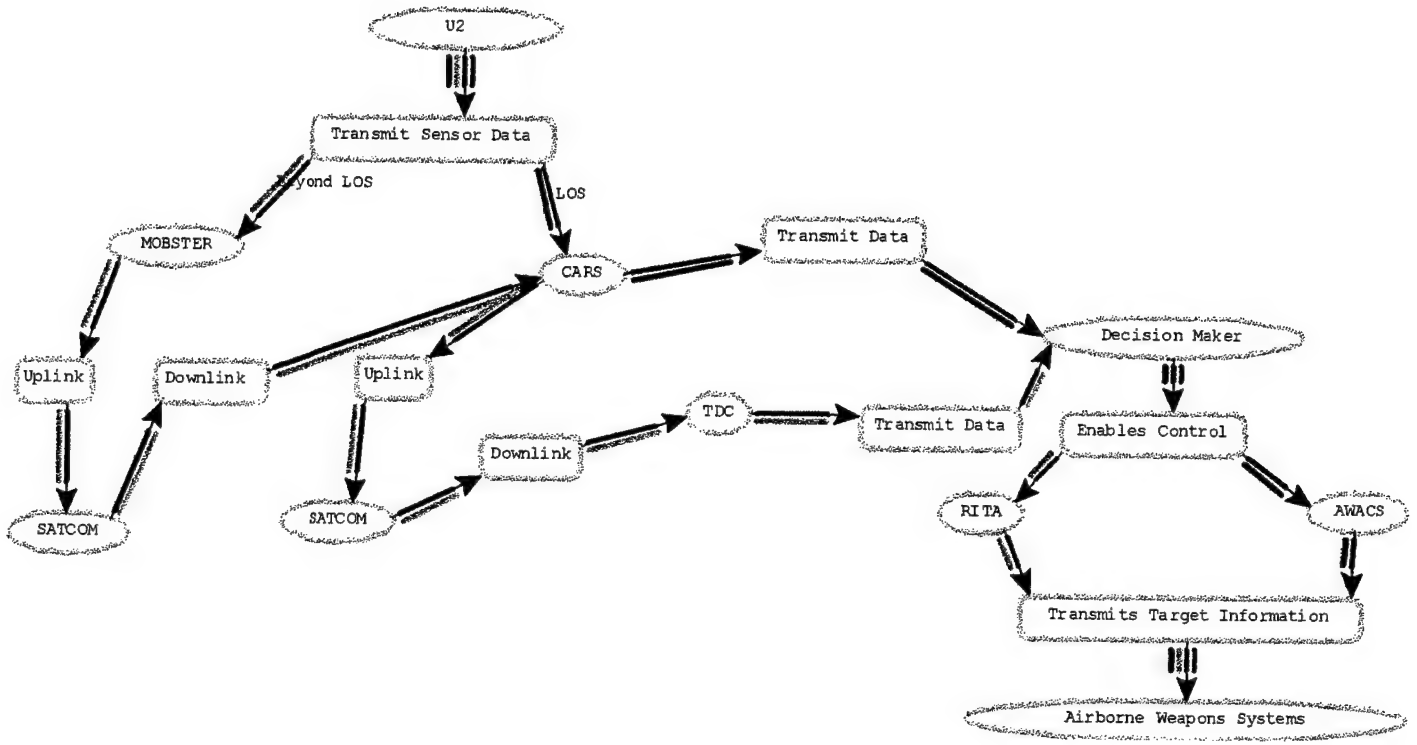


Figure 1 U2 Target Acquisition of a TBM Launcher to Attack
The four independent processes that can occur are shown using LEdit to construct a CPN model of this function.

a. A generalized notation for identifying, counting and assessing the various communication paths between communication nodes.

b. A methodology for embedding a Figure of Merit (FOM) into LEdit graphical designs. The FOM will be a metric for evaluating the efficiency of the data transmission between two nodes of the communications network.

c. Software to read the LEdit output file and obtain a quantitative description of the communications network.

d. Analysis software to determine the optimal communications path.

e. Software to provide communication-path data to THUNDER in an acceptable format.

2. Analysis of the Problem

This paragraph proposes solutions to the problems inherent in the development of ACE Link. As such, it serves as a Interface Design Document (IDD) for designing and coding the software described in items c, d and e of paragraph 1.2.

2.1 Generalized Notation

In the example of Figure 1, the individual communications nodes are each given a descriptive name; however, there is no requirement that this name be unique (e.g., the communications node SATCOM is referenced twice). This can cause confusion when analyzing the communications network. While LEdit retains the name of each node, it assigns an integer to each node as a unique identifier of it. We will follow this notation and map the communications nodes to positive integers in the range 1..N. In this way, the node "U2" is mapped to 1, "MOBSTER" to 2 and "CARS" to 3. We will reference an individual node by specifying the unique identifier encapsulated within brackets; for example: $|n>$ and $<n|$, the first notation indicating that the node is a receiver and the second indicating that it is a transmitter. The transition between the nodes "U2" and "MOBSTER" and the transition between "U2" and "CARS" are, in the notation of LEdit, both depicted by a single rectangular box. This is an acceptable notation for a qualitative description of the communications network, but fails to describe the details needed for a quantitative assessment of the network. The name "Transmit Sensor Data" is a convenient tag for indicating that data can be transmitted between the two nodes and will be retained as a summary name for the quantitative statistics describing the transmission of data between. Such details of the transition are not included in the basic LEdit model, but can be added to it to provide the information needed by THUNDER. In the most general case, there may be a large number of metrics that describe the transition, including such things as the number of bytes of data transmitted and the time delay before the data is available at the receiver node. Each of these metrics can be denoted by an operator that represents the computation of the metric's value as a function of the transmitting and receiving states. The notation we will use for this is:

$< m \mid T \mid n >$ = The value of the metric represented by the operator T for the transition from state m to state n. If there are N nodes, we can represent the operator T by an N x N matrix and use the equivalent notation $T_{m,n}$ for $< m \mid T \mid n >$. We will use the notation **T** to represent the matrix itself.

2.1.1 Application to TCT

Time Critical Targeting, as the name implies, is a function for which time delays in transmitting and receiving information can adversely affect the outcome of the attack or, in the most extreme cases, preclude even the possibility of an attack. The matrix **D** will represent the time delay, in seconds, for the transmission of data between communications nodes on the network. **D** is not to be thought of as a "state function", since we do not ascribe the delay to either the transmitting or receiving node, but rather as a "path function" by which the delay occurs as a result of the combined characteristics of the transmitter, receiver and communications path rather than simply the communications nodes themselves. **D** is not, in the most general case, symmetric. That is, $D_{i,j} \neq D_{j,i}$. That is, data transmission between two nodes is directional in the sense that the time delays are not identical for the two transmission paths. Also, the diagonal matrix elements, $D_{i,i}$, describe transmission paths that do not physically occur and therefore lead to infinite time delays. We will express this, computationally, by assigning the value -1 to both the diagonal elements of **D**, as well as to those elements which represent non-allowed data transmission paths. A transmission path can be inhibited for a number of reasons: damage due to enemy action, malfunctions inherent in the communications hardware and non-deployment of communications equipment being the three most common. In general, **D** is not static. That is, the values of the matrix elements are a function of time. Consider the following case: information is transmitted by state m and received by state n, once the communications path is established, with a time delay of 100 seconds. If this communications path is deployed 30 minutes into a tactical scenario, we would represent the matrix elements $D_{m,n}$ by the following expression:

$$D_{m,n} = -1 \qquad t \leq 1800$$
$$D_{m,n} = 100 \quad t > 1800,$$

where t represents the event time in seconds. While non-static modeling of communications time delays increases the realism of the analysis, the mechanics for arriving at the optimal path is made only slightly more difficult by this extension. At any event time, t, the matrix elements of **D** must first be evaluated before an optimal path analysis can be performed. As more realism is added to the model, the techniques that must be employed to describe the network's behavior must be extended. There are two cases that should considered:

2.1.1.1 Timed Events

A timed event is one for which a matrix element $D_{m,n}$ has values that depend, in a pre-determined way, upon the event time. This is an extension of the ideas described in paragraph 2.1.1 for a communications path that is deployed at some time after the event has started. The analyst may enter a table of values specifying the expected time delay during specified range of event times.

2.1.1.2 Stochastic Events

The assumption that the time delay for a particular communications path is constant leads to a lack of realism and often biases the results of the analysis from the optimal solution. A more robust approach is to consider the time delay to be a random variable from a specified distribution. Experimental data that describe the network are most often specified in this way. Both the occurrence of the event itself and the resulting value of the time delay can be described as random variables. The following notation can therefore be applied to the time the event occurs and to the time delay, but with different interpretations. Let x be a random variable from a distribution with mean M and variance V. The notation

$$x = P(M,V),$$

express this statement in compressed notation; P is the type of distribution as expressed by the probability distribution function (PDF). Two types of distribution will be considered: Normal (or Gaussian) and Uniform. In the former case, P=G and in the later case P = U. The extension to other types of PDFs is straightforward. Consider the following statements (hypothetical) describing a particular communications path.

a. During normal operation, the communications time can be expressed as a random sample from a Gaussian distribution whose mean is 250 seconds and whose variance is 25 seconds.

b. During degraded operation, the communications time delays can be expressed as a random sample from a Gaussian distribution whose mean is 400 seconds and whose variance is 100 seconds.

c. The communications path remains degraded for a period of time that can be expressed as a random sample from a Gaussian distribution whose mean is 100 seconds and whose variance is 20 seconds.

d. Once deployed, the communications path operates normally until degradation occurs. The time after deployment when degradation occurs can be expressed as a random sample from a Gaussian distribution whose mean is 20 hours and whose variance is 2 hours.

These four statements describe a realistic situation that occurs with all communications systems, although perhaps with different numerical values. The notation we have introduced above can be used to model this system's behavior:

$a = 3600*G(20,2)$
$b = G(100,20)$
$D_{m,n} = G(250,25) \quad t \le a$
$D_{m,n} = G(400,40) \quad a+b \ge t > a,$
$D_{m,n} = G(250,25) \quad t > a+b$

In order to implement this behavior, the times a and b will be pre-computed before the start of the event.

2.1.2 Scope of the Problem

If a network has N nodes, there are N (N-1) communications paths between the nodes. These paths are represented by the off-diagonal elements of the matrix **D**, which, for TCT, is the time delay in seconds. We would like to find the path that minimizes the total time delay for initial transmission from one node, a, to reception by a second node, b. We define the total time delay, $\mathcal{D}$, as

$$\mathcal{D} = Ta,x + Tx,y + ..Tz,b,$$

where the intermediate states along the path are denoted by x, y..z. While the path with the smallest number of transitions is the path directly from node a to node b, this path may not be allowed or it may not produce the minimum time delay. If there are N nodes in the network, there are I = N-2 possible intermediate nodes for the transmission path, and no node may occur more than once in forming the sequences of intermediate nodes that form all possible transmission paths. There are I paths with one intermediate node, I(I-1) paths with two intermediate nodes, I(I-1)(I-2) paths with three intermediate nodes, I(I-1)(I-2)(I-3) paths with four intermediate nodes. In general, if there are n intermediate nodes, the number of independent paths, P_n, is given by the formula

$$P_n = I! / (I-n)!$$

where k! is equal to the product of the positive integers from 1 to k and 0! = 1. In this equation, n is in the range 0 to I. The total number of possible paths is found by summing P_n from n=0 to n=I. The result is approximately I!e, where e is the base on the natural logarithms. Writing this result as a function of N, we get

Total Number of Possible Transmission Paths $\cong$ (N-2)!e.

For example, a network consisting of 17 nodes produces approximately 3.5×10^{12} transmission paths between any pair of nodes. This result is so large that even high-speed microprocessors would be ineffective when confronted with the task of considering each of these paths individually. As a result, it will be mandatory for the analyst to identify all of the possible transmission paths in the design stage of the problem. This is the main advantage of using LEdit, since it provides a reasonably simple way of graphically identifying the possible transmission paths which can be exploited for the purpose of finding the best of these paths.

2.1.3 Phase I Limitations

The first phase of this investigation is designed to establish a methodology for integrating LEdit and THUNDER and to demonstrate this capability in a "proof of concept" design. Therefore, the FOM for Optimal Transmission Path analysis will be limited to the simple case of a static, non-stochastic time delay for transmission by one node and reception by a second node.

2.2 Embedding Transmission Path Data in LEdit

The previous paragraph has pointed out the necessity of identifying the possible transmission paths between two nodes of the network during the design stage of the problem. It is clear that, in the types of problems that must be considered for battlefield simulations, the transmitter and receiver (that is, the initial and final nodes that define the transmission paths) are specified by the problem. Thus, a single graphical design specifying the various transmission paths between these two nodes will be sufficient for our purposes.

In order to extend the use of LEdit to include FOM data for each path segment, we must identify a general methodology for meeting the requirements of items (b) and (c) of paragraph 1.2. We want to provide an "Optimal Transmission Path" capability to the LEdit-THUNDER integration that can be achieved with a minimum of changes to the source code of these programs, and therefore we will use information that is currently available in the output file of LEdit to translate a network design into data required to perform an Optimal Transmission Path analysis. The following data must be obtained from the LEdit design:

1. Identification of each node and map it to an integer; associate the node's name with its unique identifier

2. Identification of all transitions between pairs of nodes, including the direction of data flow

3. The time delay, or other data specifying the FOM, for each transition between nodes

4. Identification of all paths from the transmitting node to the receiving node. Each path identification must included an ordered list of nodes that make up the path

2.2.1 LEdit File Format - Existing Capability

LEdit generates a text file for each network design. In this paragraph, we will indicate how the data described immediately above can be embedded in, and extracted from, this file. Figure 2 is a simple network design created using LEdit. Figure 3 contains a portion of the text file associated with this design.

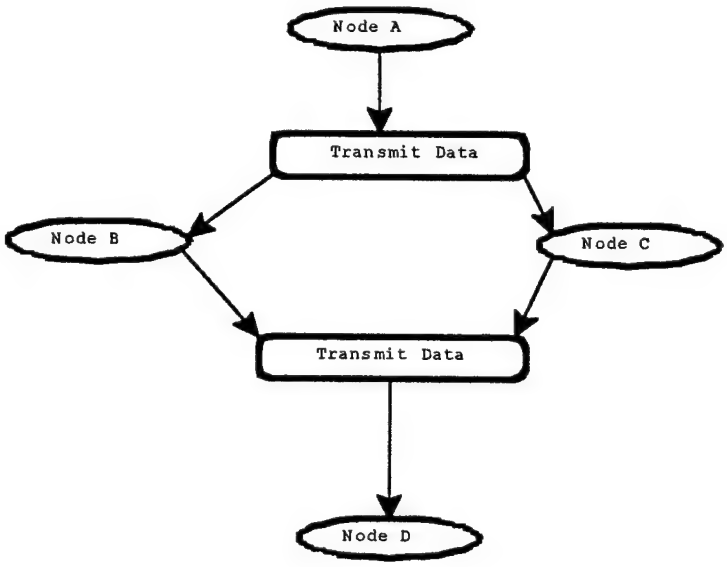


Figure 2. Simple Network Design Created with LEdit.

The network consists of four (4) nodes and four (4) transitions between nodes. The nodes are:

Node Tag	**Node Location**
Node A	Top Center
Node B	Middle Left
Node C	Middle Right
Node D	Bottom Center

The tags, or names, have been entered into the design by the following process:

(1) After placing a node in the design, double left click the mouse while the mouse cursor is on the node.

(2) A Dialog Box is displayed. Type the name of the node in the Vertex Box.

Node names are not unique identifiers, since there is no requirement that the names of the nodes must be unique. Internally, within LEdit, each node is assigned a integer that is used as the unique identifier. The association of the node's tag with the unique identifier is the first item in the specification of paragraph 2.2. There are four (4) transitions in the design.

Transition	**Transmitting/Receiving Nodes**
1	<Node A | Node B>
2	<Node A | Node C>
3	<Node B | Node D>
4	<Node C | Node D>

The transitions have been formed by the following process:

(1) Place a transmitter node, receiver node and transition in the design.

(2) Move the mouse cursor over the transmitter node, press the left mouse button and hold it down.

(3) Drag the transmitter node onto the transition icon (rectangle) and release the mouse button. An arrow will be drawn in the design in the direction of data flow from the transmitting node to the transition icon.

(4) Move the mouse cursor over the transition icon, press the left mouse button and hold it down.

(5) Drag the transition icon onto the receiver node and release the mouse button. An arrow will be drawn in the design in the direction of data flow from the transition icon to the receiver node.

The identification of these four transitions and the specification of the transmitter and receiver node for each is the second item in the specification of paragraph 2.2. We have identified the nodes and transitions by visually inspecting the graphical design of the network produced using LEdit. We would like to automate this process by obtaining the information directly from the text file produced by LEdit. This file is stored with the extension ".edt". Figure 3 shows the first several pages of the file created for the design in Figure 2. The processing steps described in Table 1 can be implemented by middleware to perform this function:

```
rmaps()
#fieldspeclists()
#looptypes(
        Generic())
#vertices(
        1:SMStateVertex.
                SMStateVertex:
                        HBGOval(
                                label:string("","Node A  "),
                                iconicWidth:integer(12,24),
                                iconicHeight:integer(12,18),
                                hasEmphasis:boolean(false,false),
                                isSelected:boolean(false,false),
                                drawableRank:integer(0,0)):
                        ParameterMap(
                                (label,label,RA),
                                (isStartingVertex,hasEmphasis,RA),
                                (isSelected,isSelected,RA),
                                (exitRank,drawableRank,RA)).
                        noName(
                                isSelected:boolean(false,false),
                                label."Vertex Label":string("","Node A  "),
                                freeText."Free Text":string("","","MaxChars:65535"),
                                isStartingVertex:boolean(false,false),
                                exitRank:integer(0,0))."",
        2:SMStateVertex.
                SMStateVertex:
                        HBGOval(
                                label:string("","Node B  "),
                                iconicWidth:integer(12,24),
                                iconicHeight:integer(12,18),
                                hasEmphasis:boolean(false,false),
                                isSelected:boolean(false,false),
                                drawableRank:integer(0,0)):
                        ParameterMap(
                                (label,label,RA),
                                (isStartingVertex,hasEmphasis,RA),
                                (isSelected,isSelected,RA),
                                (exitRank,drawableRank,RA)).
                        noName(
                                isSelected:boolean(false,false),
                                label."Vertex Label":string("","Node B  "),
                                freeText."Free Text":string("","","MaxChars:65535"),
                                isStartingVertex:boolean(false,false),
                                exitRank:integer(0,0))."",
        3:SMStateVertex.
```

```
        SMStateVertex:
                HBGOval(
                        label:string("","Node C  "),
                        iconicWidth:integer(12,24),
                        iconicHeight:integer(12,18),
                        hasEmphasis:boolean(false,false),
                        isSelected:boolean(false,false),
                        drawableRank:integer(0,0)):
                ParameterMap(
                        (label,label,RA),
                        (isStartingVertex,hasEmphasis,RA),
                        (isSelected,isSelected,RA),
                        (exitRank,drawableRank,RA)).
                noName(
                        isSelected:boolean(false,false),
                        label."Vertex Label":string("","Node C  "),
                        freeText."Free Text":string("","","MaxChars:65535"),
                        isStartingVertex:boolean(false,false),
                        exitRank:integer(0,0)).",
4:SMStateVertex.
        SMStateVertex:
                HBGOval(
                        label:string("","Node D  "),
                        iconicWidth:integer(12,24),
                        iconicHeight:integer(12,18),
                        hasEmphasis:boolean(false,false),
                        isSelected:boolean(false,false),
                        drawableRank:integer(0,0)):
                ParameterMap(
                        (label,label,RA),
                        (isStartingVertex,hasEmphasis,RA),
                        (isSelected,isSelected,RA),
                        (exitRank,drawableRank,RA)).
                noName(
                        isSelected:boolean(false,false),
                        label."Vertex Label":string("","Node D  "),
                        freeText."Free Text":string("","","MaxChars:65535"),
                        isStartingVertex:boolean(false,false),
                        exitRank:integer(0,0)).",
5:SMActionVertex.
        SMActionVertex:
                HBGRoundedBox(
                        label:string("","   Transmit Data  "),
                        iconicWidth:integer(12,24),
                        iconicHeight:integer(12,24),
                        hasEmphasis:boolean(false,false),
```

```
                    isSelected:boolean(false,false),
                    drawableRank:integer(0,0)):
                ParameterMap(
                    (label,label,RA),
                    (isStartingVertex,hasEmphasis,RA),
                    (isSelected,isSelected,RA),
                    (exitRank,drawableRank,RA)).
                noName(
                    isSelected:boolean(false,false),
                    label."Vertex Label":string("","  Transmit Data  "),
                    freeText."Free Text":string("",NULL,"MaxChars:65535"),
                    isStartingVertex:boolean(false,false),
                    exitRank:integer(0,0),
                    freqDist."Frequency Distribution":string("",""),
                    freqDistParm1."Frequency Distribution Parameter
1":real(0,0),
                    freqDistParm2."Frequency Distribution Parameter
2":real(0,0),
                    freqDistParm3."Frequency Distribution Parameter
3":real(0,0))."".""."0"."0"."0",
    6:SMActionVertex.
        SMActionVertex:
            HBGRoundedBox(
                label:string("","  Transmit Data  "),
                iconicWidth:integer(12,24),
                iconicHeight:integer(12,24),
                hasEmphasis:boolean(false,false),
                isSelected:boolean(false,false),
                drawableRank:integer(0,0)):
            ParameterMap(
                (label,label,RA),
                (isStartingVertex,hasEmphasis,RA),
                (isSelected,isSelected,RA),
                (exitRank,drawableRank,RA)).
            noName(
                isSelected:boolean(false,false),
                label."Vertex Label":string("","  Transmit Data  "),
                freeText."Free Text":string("",NULL,"MaxChars:65535"),
                isStartingVertex:boolean(false,false),
                exitRank:integer(0,0),
                freqDist."Frequency Distribution":string("",""),
                freqDistParm1."Frequency Distribution Parameter
1":real(0,0),
                freqDistParm2."Frequency Distribution Parameter
2":real(0,0),
```

```
                    freqDistParm3."Frequency Distribution Parameter
3":real(0,0))."".""."0"."0"."0")
#regions(
        1:CompoundGraphRegion:0:0[0,0][1000000,1000000],
        2:SimpleGraphRegion:1:1[216,20][84,24],
        3:SimpleGraphRegion:2:1[93,130][84,24],
        4:SimpleGraphRegion:3:1[331,131][84,24],
        5:SimpleGraphRegion:4:1[224,283][84,24],
        6:SimpleGraphRegion:5:1[210,83][119,27],
        7:SimpleGraphRegion:6:1[204,188][125,27])
#edges(
        1:SMEdge(
                sourceKey:integer(0,1),
                targetKey:integer(0,5),
                label:string("","")),
        2:SMEdge(
                sourceKey:integer(0,5),
                targetKey:integer(0,2),
                label:string("","")),
        3:SMEdge(
                sourceKey:integer(0,5),
                targetKey:integer(0,3),
                label:string("","")),
        4:SMEdge(
                sourceKey:integer(0,2),
                targetKey:integer(0,6),
                label:string("","")),
        5:SMEdge(
                sourceKey:integer(0,3),
                targetKey:integer(0,6),
                label:string("","")),
        6:SMEdge(
                sourceKey:integer(0,6),
                targetKey:integer(0,4),
                label:string("","")))
```

Figure 3 Partial Text Generated by LEdit to Describe A Network Design. The text was stored in a file with the extension .edt when the network shown in Figure 2 was designed. It contains sufficient information to obtain a list of the nodes and allowable transitions in the network. The sections of the text related to extracting node and transition data have been highlighted by the author to clarify the methodology proposed for automating this process.

Processing Step	Action	Processing/Result
1	Read line 1 of .edt file and store as string	If string ≠ "#parametermaps()" then file is corrupted - end processing with error condition
2	Read line 2 of .edt file and store as string	If string ≠ "#fieldspeclists()" then file is corrupted - end processing with error condition
3	Read line 3 of .edt file and store as string	If string ≠ "#looptypes(" then file is corrupted - end processing with error condition
4	Read line 4 of .edt file and store as string	If string ≠ "Generic ())" then file is corrupted - end processing with error condition
5	Read line 5 of .edt file and store as string	If string ≠ "#vertices (" then file is corrupted - end processing with error condition
6	Read line 6 of .edt file and store as string	If first character is not 1 then file is corrupted - end processing with error condition. If the first character is 1, the first unique identifier has been obtained.
7.	Inspect the text after 1:	If the string is "SMStateVertex.", the unique identifier 1 is mapped to a node. If the string is "SMActionVertex.", the unique identifier 1 is mapped to a transition.
8	Skip two line from the top of the vertex definition block and then read the following line (line 9) of .edt file and store as string	Extract the string after "label : string ("","" ". This string is the name associated with the node or transition
9	Store the unique identifier, vertex type and vertex name in a structure	First vertex has been extracted. Increment the vertex counter.
10	If the last identified vertex is a node, skip to line 26; if it is a transition, skip to line 30. Generally, increment the counter pointing to the next line to be read by 20 if the last vertex is a node and by 24 if it is a transition	Repeat the process described by processing steps 6, 7 and 8 to extract the next vertex. However, if the string in the first line of text is "#regions(", vertex identification is complete
11	Read all of the stored structures representing vertices and count the number of nodes (N) and the number of transitions (T).	
12	Declare a vector of N+T structures, A, the elements of	

	which are two integers representing the two vertices of an arc (vertex1, vertex2).	
13	Declare an NxN matrix of long integers, T, where N is the number of identified nodes. Set all of the matrix elements to -1	The default for all transitions is that the transition is unallowable.
14	Read each line until the string "#edges(" is encountered.	Begin counting transitions.
15	Read the next line and store it as a string	If the string ≠ "1:SMEdge(", do not increment the transition counter. No transitions are allowed. Otherwise, continue with the next processing step and increment the transition counter to n=1.
16	Read the next line	If the string is "sourcekey:integer (0,a)," extract the integer a and set the state1 component of the n-th element of the matrix A to a.
17	Read the next line	If the string is "targetkey:integer (0,b)," extract the integer b and set the state2 component of the n-th element of the matrix A to b.
18	Skip a line and read the next line	If the string ≠ "n+1:SMEdge(", do not increment the transition counter. No additional transitions are allowed. Otherwise, continue with the next processing step and increment the transition counter to n+1.
19	Read the next line	If the string is "sourcekey:integer (0,a)," extract the integer a and set the state1 component of the n+1-th element of the matrix A to a.
20	Read the next line	If the string is "targetkey:integer (0,b)," extract the integer b and set the state2 component of the n+1-th element of the matrix A to b.
21	Repeat steps 18, 19 and 20	
22	Inspect all element of the array A. If the two component of the structure are both nodes or both transitions, an error has occurred.	Stop processing and exit with an error message.

23	If all elements of A consists of a transition and node or a node and transition, consider the array elements individually by incrementing a counter from 1 to N+T.	Begin identifying allowed transitions.
23	Consider the i-th element of A and inspect the values of the structure components state1 and state2.	If the state1 component is a node, extract the state1 component a and the state2 component b. Loop over all other element of A and select those elements for which state1 = b. For each, extract the state2 component c. Set the matrix element $T_{a,c} = 0$.
24	Increment the counter for elements of A and repeat step 23 until all elements of A have been considered.	The transition matrix now consists of elements whose value is -1 (not allowed) or 0 (allowed).

Table 1 Middleware Design Specification for Extracting the Transition Matrix From the LEdit Text File. The processing steps described in this table, when implemented in software, will satisfy requirements 1 and 2 of paragraph 2.2.

2.2.2 LEdit File Format - Enhanced Capability

The current implementation of LEdit does not provide a direct way of adding FOM data to quantitatively describe communications paths. However, there is a straightforward way of doing so that meets the requirements of this investigation. Each arc of the CPN graph represents one part of a transition between two nodes: two arcs make up a transition, one from the transmitting node to the transition icon and the other from the transition icon to the receiving node. We can think of these two arcs as graphically representing the transmitting and receiving parts of the process by which information is sent between nodes. The following process is used to embed FOM data describing each transition:

a. Beginning with the CPN graph of Figure 1, move the mouse cursor over the arc connecting node 1 with Transition 1 and double click the left mouse button.

b. The Edit Edge Dialog Box is displayed. In the Label field, type in the time delay for the transmission part of the transition.

c. Move the mouse cursor over the OK button and single click the left mouse button to return to the graphical view.

d. Move the mouse cursor over the arc connecting Transition 1 with node 2 and double click the left mouse button.

e. The Edit Edge Dialog Box is displayed. In the Label field, type in the time delay for the reception part of the transition.

f Move the mouse cursor over the OK button and single click the left mouse button to return to the graphical view.

g. Repeat steps a..f for all allowable transitions between pairs of nodes.

Figure 4 shows the network depicted in Figure 2 with time delay data embedded into the design.

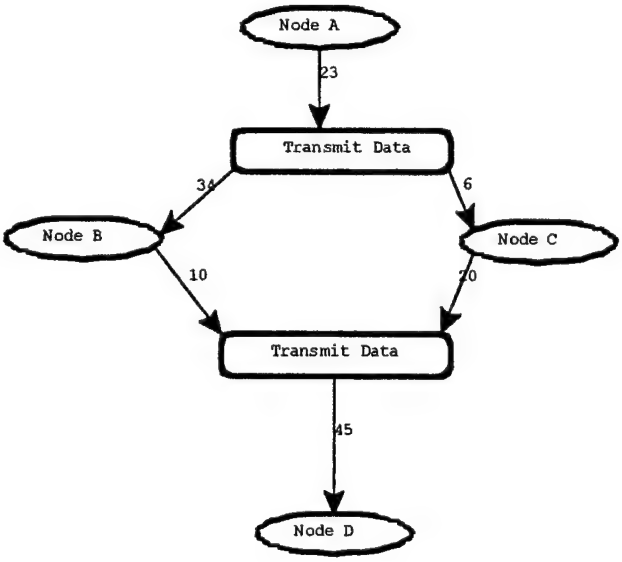


Figure 4 Communications Network with Embedded FOM Data. This network design is similar to the one depicted in Figure 2. However, we have now added time delays for each arc of the graph. Using this information, we can see that, for example, the time delay for information to be transmitted from Node A to Node B is 57 seconds.

This methodology for embedding FOM data in a graphical design has great generality and can be extended to include a range of different types of descriptive information. Furthermore, it can be extracted from the text file generated by LEdit is a way similar to that described in Table 1. The text output of the LEdit graphical design now contains the FOM data that has been added to the network design. The block of text describing the transition arcs begins with the string "#edges (". Figure 5 contains this block of text for

the CPN design in Figure 4. A comparison with the block in Figure 3 will clarify how the FOM data is embedded in the design.

```
#edges(
        1:SMEdge(
                sourceKey:integer(0,1),
                targetKey:integer(0,5),
                label:string("","23")),
        2:SMEdge(
                sourceKey:integer(0,5),
                targetKey:integer(0,2),
                label:string("","34")),
        3:SMEdge(
                sourceKey:integer(0,5),
                targetKey:integer(0,3),
                label:string("","6")),
        4:SMEdge(
                sourceKey:integer(0,2),
                targetKey:integer(0,6),
                label:string("","10")),
        5:SMEdge(
                sourceKey:integer(0,3),
                targetKey:integer(0,6),
                label:string("","20")),
        6:SMEdge(
                sourceKey:integer(0,6),
                targetKey:integer(0,4),
                label:string("","45")))
```

Figure 5 LEdit Text File when FOM Data is Embedded into a Network Design.

Table 2 contains a description of the additional processing requirements for extracting FOM data from an LEdit text file. Each step has been sequenced so that it can be integrated into the requirements described in Table1. Changes in the processing steps of Table 1 are indicated by boldface type.

Processing Step	Action	Processing/Result
12	Declare a vector of N+T structures, A, the elements of which **three integers representing the two vertices of an arc (vertex1, vertex2) and td,**	

	the time delay for that part of the transition.	
17A	**Read the next line**	**If the string is "label:string("","x")),", extract the integer x and set the td component of the n-th element of the matrix A to x.**
18	**Read the next line**	If the string ≠ "n+1:SMEdge(", do not increment the transition counter. No additional transitions are allowed. Otherwise, continue with the next processing step and increment the transition counter to n+1.
20A	**Read the next line**	**If the string is "label:string("","x")),", extract the integer x and set the td component of the n+1-th element of the matrix A to x.**
21	Repeat steps 18, 19**, 20 and 20A**	
23	Consider the i-th element of A and inspect the values of the structure components vertex1 and vertex2.	If the vertex1 component is a node, extract the vertex1 component a, the vertex2 component **b and the td component t_1**. Loop over all other element of A and select those elements for which vertex1 = b. For each, extract the vertex2 component c **and the td component t_2**. Set the matrix element $T_{a,c} = t_1 + t_2$.

Table 2 Middleware Design Specification for Extracting the Transition Matrix with Time Delay Values From the LEdit Text File. The processing steps described in this table, when implemented in software, will satisfy requirement 3 of paragraph 2.2.

2.2.3 Optimum Transmission Path

The optimum transmission path could, in principle, be calculated directly from the matrix **T**. We have shown the impracticality of doing so in paragraph 2.1.2: for any realistic network, the time required to consider all possible transmission paths is excessively large. A better approach is to use the power of LEdit to identify the possible

transmission paths in the network design. In the nomenclature of LEdit, a transmission path staring with the first node and ending with the last is referred to as a "loop". The designer can identify and color the individual arcs that comprise a loop. Figure 6 shows the communications network of Figures 2 and 4 with two loops identified. As we have shown previously, there are five possible loops. However, three of these are not allowed.

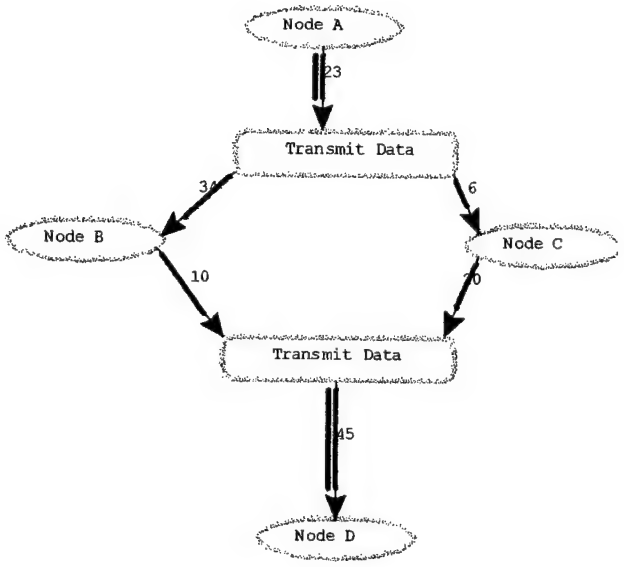


Figure 6 Communications Network with Two Loops Identified. The re loop consist of the transitions: <Node A | Node B> and <Node B | Nod D>. The delay time for this loop is $D = T_{1,2} + T_{2,4} = 57 + 55 = 11$ seconds. The blue loop consist of the transitions: <Node A | Node C and <Node C | Node D>. The delay time for this loop is $D = T_{1,3} + T3$, = 29 + 65 = 94 seconds. Clearly, the blue loop is optimal, if it i available.

2.2.3.1 Extracting Loop Specifications From LEdit Text Files

The LEdit text file contains a description of the loops that can be used to identify the two transmission paths. Figure 7 contains a segment of this text file that will be used to extract loop specifications.

```
#typedLoops(
        "Generic":{1,2,4,6}(
```

```
        Name."Loop name":string("aName","Loop 1"),
        Color."Loop color":integer(0,3),
        IsVisible:boolean(false,true),
        Flow."Flow through thread":real(0,NULL)),
"Generic":{1,3,5,6}(
        Name."Loop name":string("aName","Loop 2"),
        Color."Loop color":integer(0,7),
        IsVisible:boolean(false,true),
        Flow."Flow through thread":real(0,NULL)))
```

Figure 7 Segment of LEdit Text File Used for Loop Analysis

Table 3 describes the processing steps that will be required to identify the loops and calculate the time delay for each path.

Processing Step	Action	Processing/Result
24	Read the next line	If the line consists of the string "#typedLoops(".
25	Read the next line	If the line consists of the string " "Generic":{a,b,c..}(", extract the list of k edges and store them in the array E(n, j). Where n is the loop unique identifier and j = 1..k and the indices of the edges in the graph. Assign E(1,1) = a, E(1,2) = b, etc. until all of the edge identifiers have been used. Set E(1,0) to the number of edges in the loop. Increment the loop counter λ.
26	Skip three lines and repeat step 25 for the next loop.	The array E is completed.
27	Find the maximum of E(n,0) for n = 1.. λ	This defines the maximum number of edges in a loop, μ.
28	Declare an array L of integers with dimension $\lambda\mu$	This array will be used to store the loop node sequences.
29	Iterate over the elements of each row of the matrix E	Consider each loop edge and decompose it into a pair of vertices (a,b).
30	Extract the nodes for each loop. Let i be the loop index.	Assign the first vertex identifier of the first edge to L(i,1), the second vertex identifier of the

		second edge to L(i,2), the first vertex identifier of the third edge to L(i,3),etc. until all edges have been considered.
31	Continue until all elements of L have been defined.	The loops have been specified as a sequence of nodes.

Table 3 Middleware Design Specification for Extracting Loop Specifications. The processing steps described in this table, when implemented in software, will satisfy requirement 4 of paragraph 2.2.

3. Summary of Network Design Considerations

The processing steps described in tables 1, 2 and 3 describe a methodology for describing networks with time delays using LEdit and for extracting information from LEdit's text file to compute the Optimal Transmission Path between two nodes. The results will be stored in an array , L. The zero-th element of each row of L is the number of nodes in the loop, k, the elements of each row labeled 1..k are the unique identifiers of the nodes of the network and the K+1st element of each row is the time delay for that loop. THUNDER requires, as an input, data concerning the loop with the minimum time delay, given that a set of nodes are currently active. The proposed middleware will contain a subroutine that will identify the Optimal Transmission Path with Constraints.

3.1 The Optimal Transmission Path with Constraints

The input to this subroutine will be a one-dimensional array of integers, **I**. The length of the array will be N, the number of nodes in the network. The i-th element of this array will be set to 1 if the node is operable and 0 if the node is inoperable.

The output of the subroutine will be a one-dimensional array of integers, **O**. The length of the array will be equal to the number of nodes in the optimal transmission path +1. The zero-th element of the array will be the number of nodes in the path; the i-th element, of the array will be the unique identifier for the i-th node in the optimal transmission path; the last element of the array will be the time delay for the optimal transmission path.

3.1.1 Processing Requirements for the Optimal Transmission Path with Constraints

The array L will be used to compute the output of the subroutine. The following processing steps will be implemented:

a. Iterate over all loops using the loop index i = 1.. λ

b. Extract the number of nodes in the i-th loop from L (i,0).
c. Iterate over the unique identifiers in L describing the nodes of the loop using the index k = 1.. L (i,0)

d. L(i,k). If **I**(L(i,k)) = 0, break and consider the next loop. If **I**(L(i,k)) = 1, increment k and consider the next node in the loop by repeating step d. Continue until k = L (i,0).

e. If k = L (i,0), then all of the nodes in the path are operable.

If i=1, set MinDelay = L(1, (L(1,0)+1)) and set OptimalLoopID = 1.

If i ≠1 and L(i, (L(i,0)+1)) < MinDelay, set MinDelay = L(i, (L(i,0)+1)) and OptimalLoopID = i.

After iterating over all the loops, form the output array **O**:

O(0) = L (OptimalLoopID, 0)
O (i) = L(OptimalLoopID, i), i = 1. **O**(0).
O (**O** (i) + 1) = MinDelay

Section IV

ACE Link - An Approach to Integrating Command and Control Model Architectures (III)

Interim Report
August 15, 2000

Prepared by

Modasco Inc.
58 West Michigan Street
Orlando, Florida 32806
and
Emergent Information Technology - East
1700 Diagonal Road Suite 500
Alexandria, VA 22314

For

Department of the Air Force
Air Force Research Laboratory
Wright-Patterson Air Force Base, Ohio 45433

Under Contract
F33615-00-C-1669

Table of Contents

Executive Summary

The ACE Link software, described in Technical Reports I and II, has been written and tested. ACE Link acts as middleware, binding the graphical description of a communications network with the analytical capability of THUNDER, which models theater engagements. As such, it performs the following functions:

1. It reads a text file created by THUNDER containing the name of the LEdit design to be analyzed, the transmitter and receiver node names and a list of the names of the communications nodes that are inoperable.

2. It reads the LEdit text file and extracts a mathematically tractable description of the communications network. It then locates the communications path between the transmitter and receiver nodes that does not contain an inoperable node and produces the minimum communications time delay.

3. It writes the results of the Optimal Path Analysis to a text file which can be read by THUNDER. The text file contains the communications time delay and the sequence of node names along the optimal path from the transmitter to receiver nodes.

ACE Link is designed to execute on a modern IBM-type personal computer with minimal requirements for memory and speed.

Technical Report IV, which will be delivered in mid to late September of 2000, will provide detailed instructions for using ACE Link. It will contain a "Modeler's Guide" that describes how to embed metrics into an LEdit design so the ACE Link can determine the optimal communications path. It will also assist the modeler in establishing the input and output text files used by ACE Link and provide assistance in determining the cause of errors that can occur in using it.

2.2.1 Application Type

ACE Link was developed as a Win 32 Console Application, and the "empty project" option was selected during the new-project specification phase of the development. These options were chosen so that MIcrosoft Foundation Class (MFC) files would not be added to the source code by the development environment . ACE Link is not a Windows application in the sense that it does not make use of the MFC its hardware interfaces. It will, however, execute from the Windows operating system.

2.3 Source Code Overview

ACE Link consists of one source code file (aceLink.cpp) and one header file (aceLink.h). When compiled and linked to external libraries, these two files produce the single executable file, aceLink.exe.

2.3.1 Header File (aceLink.h)

The header file consists of 95 lines of source code organized into four sections Table 1 describes the size, in Lines of Code (LOC), and function of each section. The tabulation of the number of LOCs does not include 34 in-line comment lines.

Section	Function	Size (LOC)
1	Declaration and initialization of 18 Global Constants	18
2	Definition of 10 special data structures	38
3	Declaration and initialization of 18 Global variables	19
4	Declaration of 20 Global function prototypes	20
		95

Table 1 Organization of the Ace Link Header File

2.3.2 Source Code File (aceLink.cpp)

The source code file consists of 710 lines, including compiler/linker directives, C++ source code and textual in-line comments. It is organized into a main function and twenty (20) sub-functions. Since there is no requirement to instantiate multiple objects, the powerful Object Oriented Design features of C++ are not used. Rather, the code is monolithic and designed to provide the functional performance described in Technical Report II in a straightforward, sequential manner.

2.3.3 External Libraries

Ace Link uses functions contained in three C libraries. The three header files that reference these libraries are associated with the source code by means of include statements prior to the declaration of the main function. Table 2 describes these libraries.

Number	Header File	Function
1	string.h	Provides function to manipulate character strings.
2	iostream.h	Provides functions to allow test data to be written to the host computer monitor.
3	fstream.h	Provides functions that are used to read and write to text files on the host computer hard disk.

Table 2 Libraries Used in Ace Link.

2.3.4 Utility Files

ACE Link requires three text files for its successful execution. These files contain information that defines the communications network, the constraints on the network for determining the Optimal Communications Path and Optimal Communications Path itself. Table 3 describes these files.

Number	Header File	Description
1	*.edt	The output file of LEdit describing the graphical design of a network.
2	inputFile.txt	A file read by Ace Link at the start of execution. It contains: 1. the path and name of the LEdit file that describes the communications network 2. the name of the transmitter and receiver nodes that define the end points of the Optimal Communications Path 3. the names of all communications nodes in the network that are inoperable
3	outputFile.txt	A file created by Ace Link after it identifies the Optimal Communications Path. It contains: 1. the Time Delay for the Optimal Communications Path 2. the names of all communications nodes in the Optimal Communications Path

Table 3 Ace Link Utility Files

3. Code Listings

This paragraph contains the Ace Link source code, including the header file and source code file.

3.1 Header File Listing

```
// 1. Global Constants
const int MAX = 100; // maximum characters in a line
const int MAX_VERTICES = 251; // maximum vertices in an LEdit graph
const int MAX_EDGES = 251; // maximum edges in an LEdit graph
const int MAX_LOOPS = 251; // maximum loops in an LEdit graph
const int NUMBER_OF_HEADERS =5; // number of headers in an LEdit graph
const int MAX_CHARACTERS = 51; // maximum characters in a tag
const int MAX_PATH = 51; // maximum edges path
const int MAX_FOM = 2147483647;
//
// these are the LEdit output file headers
const char* HEADER[] = {"#parametermaps()", "#fieldspeclists()", "#looptypes(",
                                    "Generic())", "#vertices(" };
// these are the input file headers
const char* IN_HEADER [] = {"LEdit File Name:","Transmitter Node:", "Receiver Node:",
                                    "Non-Operational Nodes:" };
// these are the headers for fields
const char* NODE_HEADER = "SMStateVertex:";
const char* TRANSITION_HEADER ="SMActionVertex:";
const char* NODE_NAME_HEADER = "label:string("",";
const char* EDGE_1_HEADER = "#edges(";
const char* EDGE_2_HEADER = "SMEdge";
const char* LOOP_1_HEADER = "#typedLoops(";
const char* LOOP_2_HEADER = "Generic";
const char* IN_FILE = "\\inputFile.txt";
//
// 2. Global Type Definitions
enum vertex {none, node, transition, first};
// the type used to store the name of an element of the graph
typedef char tag[MAX_CHARACTERS];
// the structure of node data
struct vertexData {
        int index;
        tag name;
        vertex vertexType;
```

```
};
// the structure of edge data
struct edgeData {
        int index;
        int sourceNode;
        int targetNode;
        int timeDelay; };
// the structure of loop data
struct loopData {
        int sequence [MAX_PATH];
        int nodes;
        tag name;};
// the structure of loop segment data
struct pathSegmentType {
        tag startVertex;
        tag endVertex;};
// the structure of loop edges
struct commPathData {
        int fom;
        int numberOfEdges;
        pathSegmentType vertex[MAX_VERTICES];};
// the structure of loop data
struct path {
        int nodes;
        int fom;
        tag name[MAX_VERTICES];};
// the structure of input data
struct inputData {
        int numberOfNopn;
        bool status;
        tag leditFileName;
        tag transmitter;
        tag receiver;
        tag nonOpNodes[MAX_VERTICES];};
// the output structure for the optimal path
struct outputData {
        int fom;
        int numberOfNodes;
        tag nodeNames[MAX_VERTICES];
        bool status;};
//
// 3. Global Variables
bool errorStatus=false;
bool edgeTerminator=false;
bool loopTerminator=false;
int line, headerCompare;
```

```
char textline[MAX];
char* nodeNamePtr;
char* loopNamePtr;
vertex lastVertex;
// f is the stream for reading the file
fstream f;
// counters
int numberOfNodes=0;
int numberOfTransitions=0;
int numberOfVertices=0;
int numberOfEdges=0;
int numberOfLoops=0;
// the arrays/structures that store the output data
vertexData V [MAX_VERTICES];
edgeData E [MAX_EDGES];
loopData L [MAX_LOOPS];
commPathData CP [MAX_LOOPS];
path Loop [MAX_LOOPS];
//
// 4. Global Procedure Prototypes
void errorMessage (int);
void eatLines (int);
char* getName (char*);
int extractNode (char*);
int extractTimeDelay (char*);
loopData extractLoopData (char*);
char* getLoopName (char*);
int getFirstNode(int,commPathData);
int getNextNode (bool,int,int,commPathData);
inputData getInputData(void);
bool readLeditFileHeaders (tag);
outputData getOptimalPath (inputData, path[], int);
bool writeToOutputFile (outputData);
// these procedures are used for testing
void printTest1Results (inputData);
void printTest2Results (int, int,vertexData []);
void printTest3Results (int, edgeData []);
void printTest4Results (int, loopData []);
void printTest5Results (int, commPathData []);
void printTest6Results (int, path[]);
void printTest7Results (outputData);
```

3.2 Source Code Listing

```
#include <string.h>
#include <iostream.h>
```

```
#include <fstream.h>
#include "AceLink.h"

void main (void) {
	bool status = true;
	// loop index
	int k=0;
	int loopIndex, edgeIndex,edgeNumber, sourceEdge, targetEdge;
	int currentNodeIndex=0;
	int currentEdges=0;
	bool newLoop = true;
	inputData A;
	outputData optimalPath;
	//
	//
	A = getInputData(); // get data from input file
	//printTest1Results(A); // use for testing
	status = A.status;
	if (status == true)
	{
		f.open(A.leditFileName, ios::in);
		if (!f)
		{
			status = false;
			errorMessage (0);
		}
		else
		{	// check whether the headers are correct
			for (int line = 0; line < NUMBER_OF_HEADERS; line++)
			{ // check the first five lines
				if (status == true)
				{
					f.getline(textline, MAX);
					if (line == 3)
						headerCompare = strcmp(strchr(textline,
'G'),HEADER[line]);
					else
						headerCompare = strcmp (HEADER[line],
textline);
					if (headerCompare != 0)
					{
						status = false;
						errorMessage (line+1);
						break;
					}
				}
```

```
                }
        }
}

if (status == true)
// the headers are correct, so the file is not corrupted to this point
//read the vertices of the graph
{
        lastVertex = first;
        do
        {
                eatLines(1);
                f.getline(textline, MAX);
                if (strstr (textline, NODE_HEADER) != NULL)
                {
                        ++numberOfNodes;
                        ++numberOfVertices;
                        V[numberOfVertices].index = numberOfVertices;
                        V[numberOfVertices].vertexType = node;
                        lastVertex = node;
                        eatLines(1);
                        f.getline(textline, MAX);
                        strcpy (V[numberOfVertices].name,getName(textline));
                        eatLines(16);
                }
                else if (strstr(textline, TRANSITION_HEADER) !=NULL)
                {
                        ++numberOfTransitions;
                        ++numberOfVertices;
                        V[numberOfVertices].index = numberOfVertices;
                        V[numberOfVertices].vertexType = transition;
                        lastVertex = transition;
                        eatLines(1);
                        f.getline(textline, MAX);
                        strcpy (V[numberOfVertices].name,getName(textline));
                        eatLines(20);
                }
                else
                        lastVertex = none;
        }while (lastVertex != none);
        //printTest2Results (numberOfNodes, numberOfTransitions,V); // test code
        // We have now read in all the vertices (nodes and transitions)
        // skip to the top of the "edges" field
        // read line until edge header is reached
        do {
```

```
        f.getline(textline, MAX);
        headerCompare = strcmp (textline, EDGE_1_HEADER);
} while (headerCompare !=0);
// Now start counting the edges

do {
        f.getline(textline, MAX);
        if (strstr (textline, EDGE_2_HEADER) != NULL)
        {
                ++numberOfEdges;
                E[numberOfEdges].index = numberOfEdges;
                f.getline(textline, MAX);
                E[numberOfEdges].sourceNode = extractNode(textline);
                f.getline(textline, MAX);
                E[numberOfEdges].targetNode = extractNode(textline);
                f.getline(textline, MAX);
                E[numberOfEdges].timeDelay = extractTimeDelay(textline);
        }
        else
                edgeTerminator = true;
} while(edgeTerminator ==false);
// printTest3Results (numberOfEdges, E); // test code
// We have now read in all the edges
// skip to the top of the "loops" field
// read line until loop header is reached
do {
        f.getline(textline, MAX);
        headerCompare = strcmp (textline, LOOP_1_HEADER);
} while (headerCompare !=0);
// now read the loop data
do {
        f.getline(textline, MAX);
        if (strstr (textline, LOOP_2_HEADER) != NULL)
        {
                ++numberOfLoops;
                L[numberOfLoops]=extractLoopData(textline);
                f.getline(textline,MAX);
                strcpy (L[numberOfLoops].name, getLoopName(textline));
                eatLines(3);
        }
        else
                loopTerminator = true;
} while(loopTerminator ==false);
//printTest4Results (numberOfLoops, L); // test data
// All required data has been exracted from the LEdit file
f.close();
```

```
//
//
//calculate the FOM for each loop and extract the loop segments
for (loopIndex = 1;loopIndex<=numberOfLoops;loopIndex++)
{
        // loop over "loops"
        CP[loopIndex].numberOfEdges = L[loopIndex].nodes;
        for (edgeIndex = 1;edgeIndex<=L[loopIndex].nodes;edgeIndex++)
        {       // loop over the edges that make up a "loop"
                edgeNumber = L[loopIndex].sequence[edgeIndex];
                sourceEdge = E[edgeNumber].sourceNode;
                targetEdge = E[edgeNumber].targetNode;
                CP[loopIndex].fom +=E[edgeNumber].timeDelay;

strcpy(CP[loopIndex].vertex[edgeIndex].startVertex,V[sourceEdge].name);

strcpy(CP[loopIndex].vertex[edgeIndex].endVertex,V[targetEdge].name);
                }
        }
        //printTest5Results (numberOfLoops, CP);
        // next we order the loop segments and extract the nodes
        for (loopIndex = 1;loopIndex<=numberOfLoops;loopIndex++)
        {
                newLoop = true;
                // the number of edges for this loop
                currentEdges = CP[loopIndex].numberOfEdges;
                Loop[loopIndex].fom = CP[loopIndex].fom;
                Loop[loopIndex].nodes = ((currentEdges -2)/2)+2;
                // get the starting node
                currentNodeIndex = getFirstNode(currentEdges,CP[loopIndex]);

strcpy(Loop[loopIndex].name[1],CP[loopIndex].vertex[currentNodeIndex].startVer
tex);
                // get the rest of the nodes
                for
(edgeIndex=2;edgeIndex<=Loop[loopIndex].nodes;edgeIndex++)
                {
                        currentNodeIndex                                        =
getNextNode(newLoop,currentEdges,currentNodeIndex,CP[loopIndex]);

strcpy(Loop[loopIndex].name[edgeIndex],CP[loopIndex].vertex[currentNodeIndex
].endVertex);
                        newLoop = false;
                }
        }
        // printTest6Results (numberOfLoops, Loop); // test code
```

```
                // find optimal path
                optimalPath = getOptimalPath(A, Loop, numberOfLoops);
                //printTest7Results (optimalPath);
                writeToOutputFile (optimalPath);
        }
        else
                cout << "Corrupted LEdit File - cannot obtain requested result" << endl;
}

void errorMessage (int x) {
                if (x==0)
                        cout << "could not open LEdit File - stop execution." << endl;
                else
                        cout << "Line " << x << " of the  LEdit file corrupted - stop
execution" << endl;
}

void eatLines (int n) {
        for (int j = 0; j < n; j++)
                f.getline(textline, MAX);
}


char* getName (char* p) {
        char x[50];
        int leading = 0;
        int lagging = 0;
        nodeNamePtr = strchr(p, "");
        nodeNamePtr = strchr(nodeNamePtr, "");
        nodeNamePtr = strchr(nodeNamePtr, ',');
        nodeNamePtr = strchr(nodeNamePtr, "");
        nodeNamePtr = strrev(nodeNamePtr);
        nodeNamePtr = strchr(nodeNamePtr, "");
        nodeNamePtr = strrev(nodeNamePtr);
        int n = strlen(nodeNamePtr);
        // strip off quotation marks
        for (int j = 1; j < n-1; j++)
                x[j-1] = nodeNamePtr[j];
        x[n-2] = nodeNamePtr[n];
        // count leading spaces
        strcpy (nodeNamePtr, x);
        for (j=0; j<n-2; j++) {
                if (nodeNamePtr[j] == ' ')
                        ++leading;
                else
                        break;
```

```
        }
        // count lagging spaces
        for (j=n-3; j>-1; j--) {
                if (nodeNamePtr[j] == ' ')
                        ++lagging;
                else
                        break;
        }
        // test data
        //cout << "Leading spaces = " << leading << endl;
        //cout << "Lagging spaces = " << lagging << endl;
        // strip off leading and lagging spaces
        int k =0;
        for (j = leading; j < (n-(2+lagging)); j++)
        {
                x[k] = nodeNamePtr [j];
                k++;
        }
        x[k] = '\0';
        strcpy (nodeNamePtr, x);
        n = strlen(nodeNamePtr);
        //cout << n << endl;
        return nodeNamePtr;
}

int extractNode (char* p) {
        char* x;
        int node;
        x = strchr (p, ',');
        if ( x[2] == ')')
                node = x[1]-48;
        else if (x[3] == ')')
                node = 10*(x[1]-48) + (x[2]-48);
        else if (x[4] == ')')
                node = 100*(x[1]-48) + 10*(x[2]-48) + (x[3]-48);
        return node;
}

int extractTimeDelay (char* p) {
        char* x;
        int time;
        x = strchr (p, ',');
        if ( x[2] == "")
                time = 0;
        else if (x[3] == "")
                time = (x[2]-48);
```

```
        else if (x[4] == "")
                time = 10*(x[2]-48) + (x[3]-48);
        else if (x[5] == "")
                time = 100*(x[2]-48) + 10*(x[3]-48) + (x[4]-48);
        else if (x[6] == "")
                time = 1000*(x[2]-48) + 100*(x[3]-48) + 10*(x[4]-48) +(x[5]-48);
        else if (x[7] == "")
                time = 10000*(x[2]-48) + 1000*(x[3]-48) + 100*(x[4]-48) + 10*(x[5]-48)
+(x[6]-48);
        else
                time = 1000000;
        return time;
}

loopData extractLoopData (char* p) {
        // local variables
        bool corruptedFile = false;
        bool commaReached = false;
        int digitCounter = 1;
        int positionCounter = 1;
        int numberOfNodes = 0;
        int node=0;
        int digit=0;
        loopData loop;
        char* x;
        int n=0;
        int digits[100][5];
        int i, j, k;
        int powers[] = {1, 10, 100, 1000};
        // read to the start of the loop definition
        // preceded by "{"
        x = strchr (p, '{');
        n = strlen(x);
        // test code follows
        //cout << x << "  " << n << endl;
        // look at each character
        if( (x[0] != '{') || (x[n-2] !='}') ) {
                cout << "Corrupted File - stop processing" << endl;
                corruptedFile = true;
        }
        else
        {
                // look at the individual elements of the loop description
                for (i=1; i<n-1; i++)
                {
                        if (x[i] ==',' || x[i] == '}')
```

```
                {
                        // digit is complete
                        // complete digit
                        digits[digitCounter][0] = positionCounter-1;
                        positionCounter =1;
                        ++digitCounter;
                }
                else if (x[i] > 47 && x[i] <58)
                {
                        // numerical digit
                        digits[digitCounter][positionCounter] = x[i]-48;
                        ++positionCounter;
                }
                else
                {
                        cout << "Corrupted File - stop processing" << endl;
                        corruptedFile = true;
                        break;
                }
        }
        }
        if (corruptedFile == false) {
                // use stored digits to identify nodes
                numberOfNodes = digitCounter-1;
                //test code follows
                //cout << numberOfNodes << endl;
                //cout << digits[1][0] <<" " << digits[2][0] << " "<<digits[3][0] <<"
"<<digits[4][0]<<endl;
                //cout << endl;
                for (j=1; j<(numberOfNodes+1); j++)
                {
                        //convert the digits to an integer
                        node =0;
                        digit = digits[j][0];
                        for (k = digit; k>0; k--)
                                node += powers[digit-k]*digits[j][k];
                        loop.sequence[j] = node;
                        //cout << " "<< loop.sequence[j];
                }
        // test code follows
        //cout << endl;
        // extract the loop name

        }
        loop.nodes = numberOfNodes;
        return loop;
```

```
}

char* getLoopName (char* p) {
        int n=0;
        int i=0;
        char x[50];
        // strip off everything before the comma
        loopNamePtr = strchr(p,',');
        //cout << p << "  "<<loopNamePtr << endl;
        // get the string length
        n=strlen(loopNamePtr);
        //cout << n << endl;
        // there are two unwanted characters at the start and three at the end
        for (i=0; i<n-5; i++)
                x[i] = loopNamePtr[i+2];
        x[n-5] = '\0';
        //cout << x << endl;
        strcpy (loopNamePtr,x);
        return loopNamePtr;
}

int getFirstNode(int n, commPathData A){
        int i,j=0;
        int counter = 0;
        int index = 0;
        char x [MAX_CHARACTERS];
        // find which tag only occurs as a start node tag
        for (i=1;i<=n;i++)
        {
                // get the test node
                strcpy(x,A.vertex[i].startVertex);
                // set counter to 0
                counter=0;
                // loop over all edges and look only at starting nodes
                for (j=1;j<=n;j++)
                {
                        if (strcmp(x,A.vertex[j].startVertex)==0)
                                ++counter;
                }
                if (counter == 1)
                {
                        index=i;
                        break;
                }
        }
        //cout << A.vertex[index].startVertex<< endl;
```

```
        return index;
}

int getNextNode (bool newLoop, int n, int m, commPathData A){
        // n is the number of vertices
        // m is the index of the previous node
        // A is the loop structure
        // remember the previous node type
        static bool first = true;
        // the index of the next node
        int index, i = 0;
        if (newLoop==true)
                first=true;
        //cout << "new loop ="<< newLoop << endl;
        for (i=1;i<=n;i++)
        {// loop over all edges
                        if (strcmp (A.vertex[i].startVertex,A.vertex[m].endVertex)==0)
                                break;
        }
        index =i;

        if (first == false)
        {
                for (i=1;i<=n;i++)
                {// loop over all edges
                        if (strcmp (A.vertex[i].startVertex,A.vertex[index].endVertex)==0)
                                break;
                }
                index = i;
        }
        first = false;
        return index;
}

inputData getInputData(void) {
        int counter = 0;
        inputData A;
        // the input stream
        fstream f;
        A.status = true;
        f.open(IN_FILE,ios::in);
        if (!f)
        {
                cout << "Cannot open the input file - program terminated (Error Code
1)."<<endl;
                A.status = false;
```

```
        }
        else
        {// read the first line of the file (Header)
                f.getline(textline,MAX);
                if(strcmp(textline,IN_HEADER[0])!= 0)
                {
                        A.status = false;
                        cout << "Input file corrupted - program terminated (Error Code
1.1)."<<endl;
                }
                else
                {// read the second line of the file (LEdit file name)
                        f.getline(textline,MAX);
                        strcpy(A.leditFileName,textline);
                        // read the third line of the file (Header)
                        f.getline(textline,MAX);
                        if(strcmp(textline,IN_HEADER[1])!= 0)
                        {
                                A.status = false;
                                cout << "Input file corrupted - program terminated (Error
Code 1.2)."<<endl;
                        }
                        else
                        {// read the fourth line of the file (Transmitter name)
                                f.getline(textline,MAX);
                                strcpy(A.transmitter,textline);
                                // read the fifth line of the file (Header)
                                f.getline(textline,MAX);
                                if(strcmp(textline,IN_HEADER[2])!= 0)
                                {
                                        A.status = false;
                                        cout << "Input file corrupted - program terminated
(Error Code 1.3)."<<endl;
                                }
                                else
                                {// read the sixth line of the file (Receiver name)
                                        f.getline(textline,MAX);
                                        strcpy(A.receiver,textline);
                                        // read the seventh line of the file (Header)
                                        f.getline(textline,MAX);
                                        if(strcmp(textline,IN_HEADER[3])!= 0)
                                        {
                                                A.status = false;
                                                cout << "Input file corrupted - program
terminated (Error Code 1.4)."<<endl;
                                        }
```

```
                                else
                                {
                                // read the non-operational nodes
                                        while (!f.eof())
                                        {
                                                f.getline(textline,MAX);
                                                ++counter;
                                                strcpy(A.nonOpNodes[counter],textline);
                                        }
                                        A.numberOfNopn = counter;
                                }
                        }
                }
        }
        f.close();
        return A;
}

void printTest1Results (inputData A) {
        if (A.status = true)
                cout << "Read Status     =  " << "Good" << endl;
        else
                cout << "Read Status     =  " << "Bad" << endl;
        cout << "LEdit File Name  =  " << A.leditFileName << endl;
        cout << "Transmitter Node =  " << A.transmitter << endl;
        cout << "Receiver Node    =  " << A.receiver << endl;
        for (int i=1; i<=A.numberOfNopn; i++)
                cout <<"Non-Operational Node #"<<i<<"  = "<<A.nonOpNodes[i]<<endl;
}

void printTest2Results (int n, int t,vertexData A[]) {
        int z = n + t + 1; // vertices start at index 1
        cout << "Nodes       = " << n << endl;
        cout << "Transitions = " << t << endl;
        cout << " Index  Type  Name" << endl;
        for (int k = 1; k < z; k++)
        {
                        cout << "  " << A[k].index << "      ";
                        cout << A[k].vertexType << "   ";
                        cout << A[k].name << endl;
        }
}
void printTest3Results (int n, edgeData A[]) {
        ++n; // required since edges start at index 1
        cout << " Index  Source Node  Target Node  Time Delay" << endl;
```

```
        for (int k = 1; k < n; k++)
        {
                cout << "  " << E[k].index << "        ";
                cout << E[k].sourceNode << "              ";
                cout << E[k].targetNode << "        ";
                cout << E[k].timeDelay << endl;
        }
}

void printTest4Results (int n, loopData A[]){
        int segments;
        // required since loops and segments start at index 1
        int loops = n+1;
        cout<<"Index "<<"Trans ";
        for (int k=1; k<loops; k++)
                cout<<" N("<<k<<")";
        cout <<"  Loop Name"<< endl;
        for (k=1; k<loops; k++)
        {
                segments =A[k].nodes + 1;
                cout<<"  "<<k<<"     "<<A[k].nodes;
                for (int j=1; j<segments; j++)
                        cout<<"   "<<L[k].sequence[j];
                cout <<"   "<<A[k].name << endl;
        }
}

void printTest5Results (int n, commPathData A[]) {
        int segments;
        int m = n+1; // loops begin with index 1
        int i,j;
        cout <<"Index  "<<"FOM   ";
        for (i=1; i<10; i++)
                cout << "   E(" << i <<")";
        cout << endl;
        for (i=1; i<m; i++)
        {
                cout <<i<<"     "<<A[i].fom << "    ";
                segments = A[i].numberOfEdges +1;
                if (segments > 11)
                        segments = 11;
                for (j=1;j<segments; j++)
                {

                        cout<<"  "<<A[i].vertex[j].startVertex<<",";
                        cout<<A[i].vertex[j].endVertex;
```

```
            }
            cout<<endl;
        }
}

void printTest6Results (int n, path A[]){
        n++; // loops are indexed starting with 1
        int i,j;
        int nodes=0;
        cout<<"Index "<<"Nodes "<<"FOM  ";
        for (i=1; i<11; i++)
                cout << "  N(" << i <<")";
        cout <<endl;
        for (i=1; i<n; i++)
        {
                cout<<i<<"      "<<A[i].nodes<<"   "<<A[i].fom;
                nodes = A[i].nodes + 1;
                for (j=1; j<nodes; j++)
                        cout<<"    "<<A[i].name[j];
                cout <<endl;
        }
}


outputData getOptimalPath (inputData A, path P[], int n){
        outputData q;
        int stringComp1, stringComp2; // results for name comparisons
        int m=n+1; // loops start with index 1
        bool conditionsMet = false;
        int nodes=0;
        int testIndex = 0;
        int testFOM = MAX_FOM; // largest 4 byte integer
        // consider each loop and see if it is the optimal one
        // for a loop to be the optimal one the following conditions must be met:
        //1. The first node in the loop must be the transmitter loop entered in the input file
        //2. The second node in the loop must be the receiver loop entered in the input
file
        //3. None of the nodes in the loop can be non-operational
        //4. Of the loops meeting conditions 1, 2 and 3, it must have the minimum value
of the fom
        //** note that such a loop may not exist
        q.status = false; // assume no such loop exists
        // loop over all the loops and find the index of the optimal path
        for (int i=1; i<m; i++)
        {
                nodes = P[i].nodes;
```

```
            // set conditionsMet to false at the start
            conditionsMet = false;
            // test conditions 1 and 2
            stringComp1 = strcmp(A.transmitter, P[i].name[1]);
            stringComp2 = strcmp(A.receiver, P[i].name[nodes]);
            if (stringComp1 == 0 && stringComp2 == 0)
                  conditionsMet = true;
            // go no farther if conditions 1 and 2 are not met
            if (conditionsMet == true)
            {
                  // set up a double loop to compare pairs of nodes
                  for (int j=1; j<= nodes; j++)
                  {
                        for (int k=1; k<=A.numberOfNopn; k++)
                        {
                              // compare nodes
                              if (strcmp(A.nonOpNodes[k],P[i].name[j])==0)
                              {
                                    conditionsMet = false;
                                    break;
                              }
                        }
                        if (conditionsMet == false)
                              break;
                  }
                  if (conditionsMet == true) // conditions 1,2 and 3 are met
                  {
                        if ( P[i].fom < testFOM )
                        {
                              testFOM = P[i].fom;
                              testIndex = i;
                        }
                  }
            }
      }

      if (testFOM==MAX_FOM && testIndex==0)
            q.status = false;
      else
            q.status = true;
      q.fom = testFOM;
      q.numberOfNodes = P[testIndex].nodes;
      for (int a=1; a<=P[testIndex].nodes; a++)
            strcpy(q.nodeNames[a],P[testIndex].name[a]);
      return q;
}
```

```
void printTest7Results (outputData A) {

        cout << "Figure of Merit = " << A.fom << endl;
        for (int i=1; i<=A.numberOfNodes; i++)
                cout << "N(" << i << ")  ";
        cout << endl;
        for (i=1; i<=A.numberOfNodes; i++)
                cout << A.nodeNames[i]<< "     ";
        cout << endl;
}


bool writeToOutputFile (outputData A){
        // instantiate a stream
        fstream fout;
        bool writeStatus = false;
        // open the output file to write
        fout.open("\\outputFile.txt", ios::out);
        if (!fout)
        {
                writeStatus = false;
                cout << "Could not write to output file" << endl;
        }
        else
        {
                if (A.status == false)
                        fout << "No Path Found" << endl;
                else
                {
                        fout << "Minimum Time Delay = " << A.fom << endl;
                        for (int i=1; i<=A.numberOfNodes; i++)
                                fout << A.nodeNames[i] << endl;
                        writeStatus = true;
                }
        }
        return writeStatus;
}
```

4. Test Case

This paragraph describes one of the test cases used to validate the performance of Ace Link. Figure 1 depicts a communications network designed using LEdit. Embedded in the design are communications time delays for each transition in the network. Four communications loops are identified and colored red, green, blue and orange. Figure 2

shows the input file used to describe and constrain the Optimal Communications Path. It states that the network to be analyzed is contained in the file test1.edt, which is located

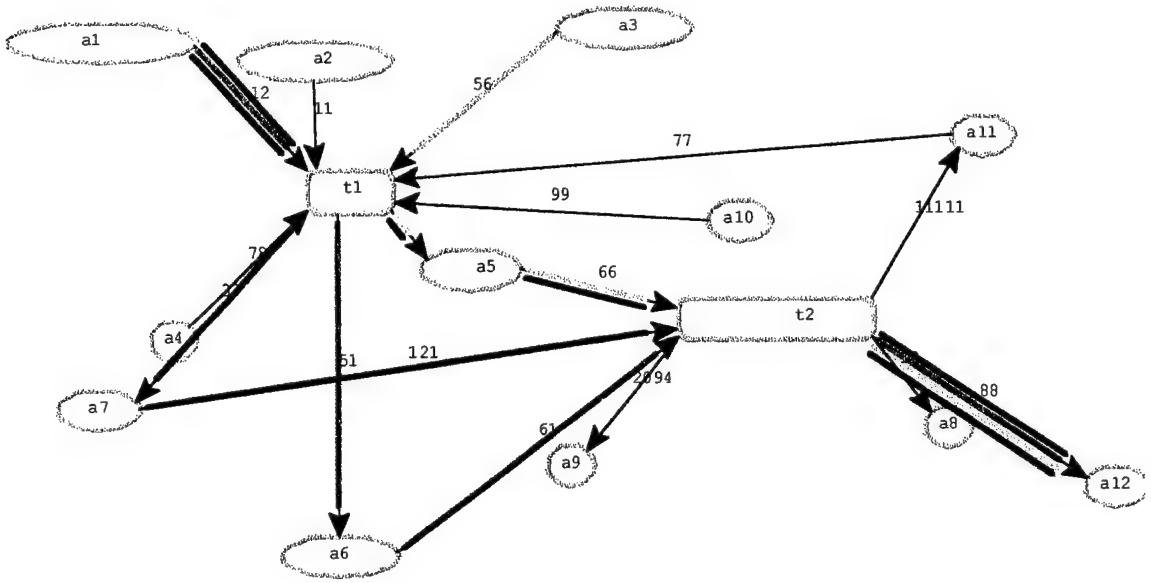


Figure 1 Test Communications Network. This network was designed using LEdit. It consists of twelve (12) communications nodes, named a1, a2..a12, and four (4) communications paths (loops) colored red, green blue and orange. LEdit stored a description of this network in a text file named test1.edt. This file is read by Ace Link to obtain the Optimal Communications Path.

LEdit File Name:
\test1.edt
Transmitter Node:
a1
Receiver Node:
a12
Non-Operational Nodes:
a2
a11
a3

Figure 2 Ace Link Input File. This file is used to describe and constrain the Optimal Communications Path for the test case.

in the root directory of the host computer's hard disk. It defines the communications path as starting at node a1 and terminating at node a12. It constrans the network by making communications nodes a2, a11 and a3 inoperable. Of the four paths (loops) identified in Figure 1, only the red, green and blue loops start at node a1 and terminate at node a12. Table 4 describes these three loops.

Loop Color	Node Sequence	Time Delay (sec)
Red	a1→a5→a12	166
Blue	a1→a7→a12	254
Green	a1→a6→a12	210

Table 4 Communication Paths in Test Network

Of these three candidates, the red loop produces the minimum time delay (166 seconds) between nodes a1 and a12. Figure 3 shows the output file generated by Ace Link. This file contains the correct description of the Optimal Communications Path and thus partially validates the Ace Link middleware.

Minimum Time Delay = 166
a1
a5
a12

Figure 2 Ace Link Output File. This file identies the Optimal Communications Path for the test case.

Section V

ACE Link - An Approach to Integrating Command and Control Model Architectures (IV)

Interim Report
September 15, 2000

Prepared by

Modasco Inc.
58 West Michigan Street
Orlando, Florida 32806
and
Emergent Information Technology - East
1700 Diagonal Road Suite 500
Alexandria, VA 22314

For

Department of the Air Force
Air Force Research Laboratory
Wright-Patterson Air Force Base, Ohio 45433

Under Contract
F33615-00-C-1669

Table of Contents

Executive Summary

Previous investigations, the results of which were described in Technical Reports I, II and III, have resulted in the development of ACE Link, a computer program that acts as "middleware" connecting LEDIT and THUNDER. This report serves as a User's Guide to ACE Link, providing detailed procedures for integrating a communications network, designed graphically using LEdit, with the battlefield simulation capability of THUNDER.

ACE Link was developed explicitly to obtain the optimal communications path as defined in paragraph 1. It can be modified in a straightforward way to perform a similar analysis for an alternate optimization prescription.

2. Introduction to ACE Link

While ACE Link is the middleware that binds two existing programs, LEdit and THUNDER, together, it is not sufficient to describe the detailed procedures for using it without also discussing how to embed time delay data in an LEdit graphical design. For this reason, we will introduce four topics:

- How to Embed TCT Data in LEdit
- How to Construct an ACE Link Input File
- How to Interpret an ACE Link Output File
- How to Execute ACE Link

2.1 Detailed Operating Procedures

The following four paragraphs describe the details for using ACE Link to determine the optimal communications path based upon the minimization of the communications time delay.

2.1.1 Embedding TCT Data in LEdit

There are five rules that must be observed when developing a communications network that can analyzed by ACE Link. These are:

Number	Rule Description	Purpose
1	All vertices (places and transitions) must be given a unique name, which must be less than 50 characters long.	This rule is established so that one can unambiguously identify nodes by their names. The limit of 50 characters is based upon the design of ACE Link; in principle, much shorter names are preferred since it reduces the clutter of the graph.
2	Each valid communications path between node pairs must be designated as a "loop" .	Establishes the possible ways communications can occur between nodes.

3	Communications between a pair of nodes is represented in LEdit by two edges. The total time delay for the communications to occur can be assigned partially to each edge, or in total to either edge.	Allows the analyst to model a transmission part of the delay and a reception part of the delay.
4	The time delay assigned to an edge must be entered in the Label field on the Edit edge dialog box, which is displayed after double left-clicking on the mouse while the cursor is placed on the edge.	Defines a simple method of embedding data in LEdit to describe a communications path.
5	The upper limit for the time delay entered for an edge is 1000000. Blank spaces and non-numerical text, including decimal points are not allowed.	Defines the format for extracting time delay values from the LEdit design.

2.1.2 Constructing an ACE Link Input File

The ACE Link input file is named inputFile.txt. It must be stored in the root directory of the host computer's hard disk. It can be easily created using any standard text editor such as UltraEdit-32. A line by line description of this file follows:

Line Number	Contents	Purpose
1	This line contains a single header statement : LEdit File Name: This line must be present or ACE Link will fail to execute.	Used to identify the line that follows as containing the name of the LEdit file that ACE Link should analyze to extract the optimum communications path.
2	This line contains the name and location of the LEdit file that ACE Link will read. The file name, including path, must be in stated in the logical file naming convention. A file name provided in the Universal Naming Convention (UNC) format will result in an error. If, for example, the LEdit file is in the root directory and named myTest.edt , either of the following entries will be sufficient:	ACE Link will open and read the contents of this file.

	\myTest.edt or c:\myTest.edt However, \\myTest.edt will result in an error when there is an attempt to open the file.	
3	This line contains a single header statement : Transmitter Node: This line must be present or ACE Link will fail to execute.	Used to identify that the following line contains the name of the transmitter node that defines the start of the optimum communications path
4	The name of the transmitter node. For proper execution, it must be the name of a vertex of the LEdit graph and the start point of an LEdit loop.	Defines the start of the communications path.
5	This line contains a single header statement : Receiver Node: This line must be present or ACE Link will fail to execute.	Used to identify that the following line contains the name of the receiver node that defines the end of the optimum communications path
6	The name of the receiver node. For proper execution, it must be the name of a vertex of the LEdit graph and the end point of an LEdit loop.	Defines the end of the communications path.
7	This line contains a single header statement : Non-Operational Nodes: This line must be present or ACE Link will fail to execute.	Used to identify that the following lines contain the name(s) of the communications node(s) that are not operational.
8..N	The name of a non-operational node. The last entry must not include a carriage return.	Defines the nodes of the LEdit graph which are not operational. Any communications path containing a non-operational node does not qualify as the optimum communications path

2.1.3 Interpreting an ACE Link Output File

The ACE Link output file is named outputFile.txt. It must be stored in the root directory of the host computer's hard disk. It can be easily created using any standard text editor such as UltraEdit-32. A line by line description of this file follows:

Line Number	Contents	Purpose
1	This line contains a header statement: Minimum Time Delay = n, where n is an integer representing the time delay in seconds for the optimum communications path.	Displays the time delay for the optimum communications path.
2	This line contains the name of the transmitter node. It will be identical to line 4 of the input file.	Identifies the first node of the optimum communications path.
3..(N-1)	These lines contain the names of the intermediate nodes in the optimum communications path	Identifies the sequence of nodes, starting with the second and ending with the next to last, that forms the optimum communications path
N	This line contains the name of the receiver node. It will be identical to line 6 of the input file.	Identifies the last node of the optimum communications path.

2.1.4 Executing ACE Link

The ACE Link executable performs the required analysis of the communications network designed in LEdit very rapidly. The computation time depends upon the host computer's speed, but, for all modern computers, the time is less than one second for even complex networks of 100 or more loops. For test purposes, ACE Link terminates upon user command after writing to the output file and then the host computer screen. This final screen display is intended to provide feedback to the user and help locate errors in either the input file, output file or LEdit file.

Successful completion, without errors, will terminate after the following messages are displayed on the screen:

Analysis Complete
Enter 0 to quit

After the user presses the 0 key and then the ENTER key, execution of ACE Link terminates.

Unsuccessful completion will terminate in two ways:

a. If a non-specific error occurs, the following messages are displayed on the screen:

Corrupted LEdit File - cannot obtain requested result
Enter 0 to quit

After the user presses the 0 key and then the ENTER key, execution of ACE Link terminates.

b. If a specific format error occurs, an error-identification statement will precede these two lines. The error-identification statements are:

Error-Identification Statement	Possible Cause
Could not open the LEdit File - stop execution.	The LEdit file could not be opened. The most likely cause is that the path/file name in inputFile.txt is incorrect.
Line n of the LEdit File is corrupted - stop execution (n = 1..5)	The header text on line n of the LEdit file is incorrect. Either the file has been corrupted or the path/file name in inputFile.txt points to a non LEdit output file.
Corrupted File - stop processing	The time delay has been incorrectly entered for an edge. It either contains a non-numeric character or is out of range.